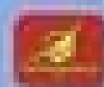
Qt应用编程系列丛书

霍亚飞 编著
吴 迪 白建平 董世明 审校

Qt Creator快速入门

- 全新。
- 经典。
- 基础。
- 系统。



北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

Table of Contents

Qt 快速入门系列教程

> 作者：yafeilinux
>
> 来源：http://bbs.qter.org/forum.php?mod=viewthread&tid=193

导语

该系列教程是基于QtCreator开发环境的Qt入门级教程。自2009年10月至今的两年多时间里，该系列教程逐渐完善，已经包含了Qt基础、2D绘图、数据库和XML、网络编程、Qt Quick等最基本和最常用的知识点。从该系列教程中衍生出的Qt专题教程和Qt系列开源软件，分别对特定应用领域进行了综合的讲解和应用。现在，该系列教程的访问量已经超过百万，基于该系列教程的《Qt Creator快速入门》和《Qt及Qt Quick开发实战精解》两本书籍已经出版。

为了便于大家更好的学习和交流，将所有教程从作者的博客网站www.yafeilinux.com全部转移到了Qter论坛（Qter开源社区www.qter.org），并将所有内容基于最新版本的Qt重新编辑整理。今后，教程的内容将会得到进一步的扩展和更新，并会在第一时间推出 Qt 5 的内容，将尽全力为广大Qt初学者提供一套易学、详尽、新颖的Qt教程

第一部分 学习Qt必备知识

（基于Qt 4，主要讲解Widget C++编程）

基础篇

第0篇 开始学习Qt 与Qt Creator

导语

自从2009年十月我在博客上写了第一篇QtCreator系列教程到现在，断断续续一共写了四十八篇，涵盖了Qt基础、绘图、数据库、Qt Quick和网络等主要应用方面的内容。虽然其中的内容很基础，但这也正是入门的读者所想要的，现在这个系列的读者已经超过了3万，很感谢大家对我的支持。因为当时开始写教程时并没有想得那么系统，所以就成了想到哪写哪。在现在看来，上来第一篇就是helloworld的编写，从来没有涉及Qt的介绍，感觉这对于一个Qt系列的教程来说是个缺陷。所以，现在我补上了这第零篇，来对Qt和Qt Creator进行一个大体上的介绍，其实，下面的内容都是整理自Qt官方网站。

因为Qt Creator系列教程好像有半年没有更新了，很多网友问我还会不会写下去。这里告诉大家，还是以前的话，我们的教程会一直写下去的，直到什么时候呢？或许会持续到Qt已经没落到无人问津的时候吧！

最后再次感谢大家对该系列教程和我们yafeilinux.com网站的支持，谢谢大家！

-------------------yafeilinux 2011-2-23

更新信息：2012年5月，基于该系列教程的系列书籍已经出版，查看详情！

Qt官方信息

- Qt官网：http://qt.digia.com/
- Qt开源官网：http://qt-project.org/
- Qt最新版本下载：http://qt-project.org/downloads
- Qt所有版本下载：ftp://ftp.qt-project.org/qt/source/
- Qt Creator所有版本下载：ftp://ftp.qt-project.org/qtcreator/

目录

正文

一、Qt与Qt Creator简介



Qt是一个跨平台应用程序和 UI 开发框架。使用 Qt 您只需一次性开发应用程序，无须重新编写源代码，便可跨不同桌面和嵌入式操作系统部署这些应用程序。

Qt Creator 是全新的跨平台Qt IDE，可单独使用，也可与 Qt 库和开发工具组成 一套完整的SDK. 其中包括：高级 C++ 代码编辑器，项目和生成管理工具，集成的上下文相关的帮助系统，图形化调试器，代码管理和浏览工具。

二、Qt功能与特性

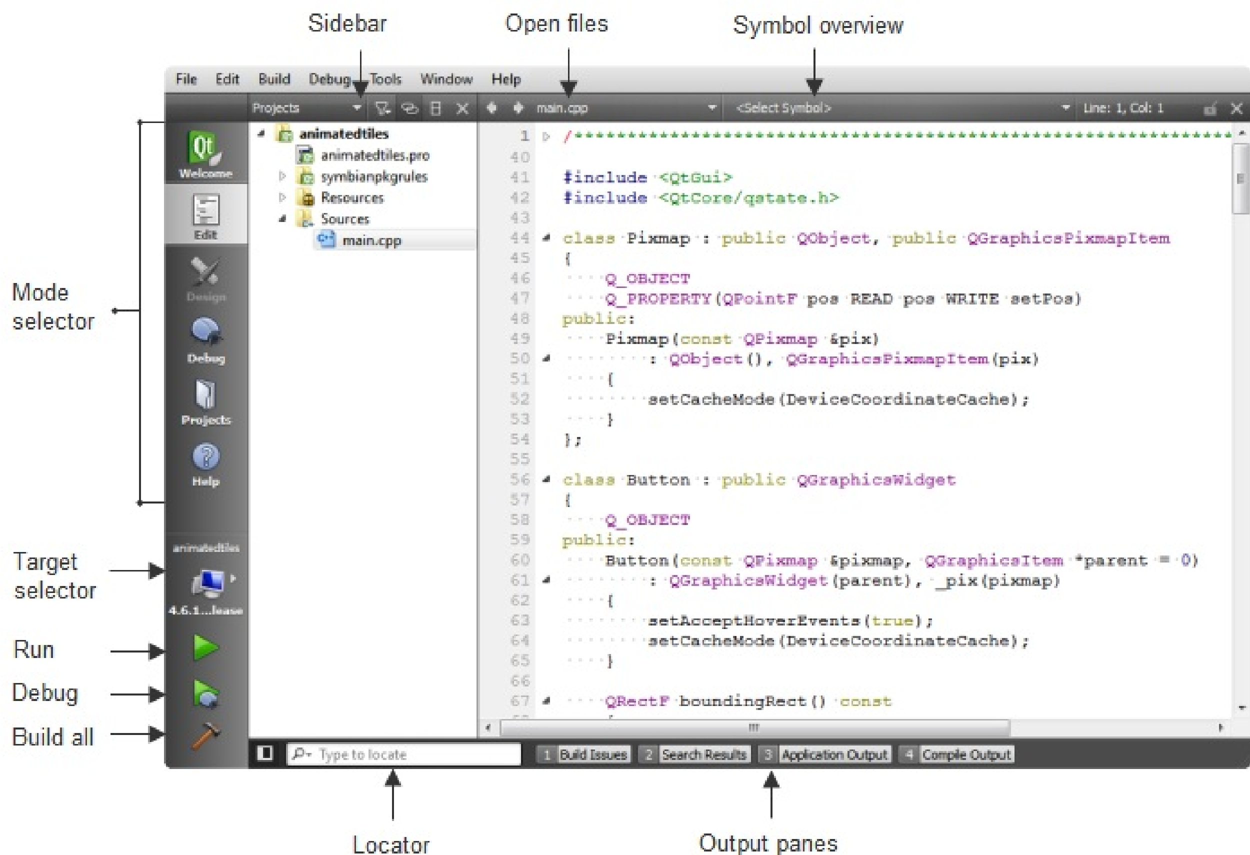


- 直观的 C++ 类库：模块化 Qt C++ 类库提供一套丰富的应用程序生成块 (block)，包含了构建高级跨平台应用程序所需的全部功能。具有直观，易学、易用，生成好理解、易维护的代码等特点。
- 跨桌面和嵌入式操作系统的移植性：使用 Qt，您只需一次性开发应用程序，就可跨不同桌面和嵌入式操作系统进行部署，而无须重新编写源代码，可以说Qt无处不在（QtEverywhere）。
- 使用单一的源代码库定位多个操作系统；
- 通过重新利用代码可将代码跨设备进行部署；
- 无须考虑平台，可重新分配开发资源；
- 代码不受担忧平台更改影响的长远考虑 ；
- 使开发人员专注于构建软件的核心价值，而不是维护 API 。
- 具有跨平台 IDE 的集成开发工具：Qt Creator 是专为满足 Qt 开发人员需求而量身定制的跨平台集成开发环境 (IDE)。Qt Creator 可在 Windows、Linux/X11 和 Mac OS X 桌面操作系统上运行，供开发人员针对多个桌面和移动设备平台创建应用程序。
- 在嵌入式系统上的高运行时间性能，占用资源少。

三、Qt Creator功能和特性



- 复杂代码编辑器：Qt Creator 的高级代码编辑器支持编辑 C++ 和 QML (JavaScript)、上下文相关帮助、代码完成功能、本机代码转化及其他功能。
- 版本控制：Qt Creator 汇集了最流行的版本控制系统，包括 Git、Subversion、Perforce、CVS 和 Mercurial。
- 集成用户界面设计器：Qt Creator 提供了两个集成的可视化编辑器：用于通过 Qt widget 生成用户界面的 Qt Designer，以及用于通过 QML 语言开发动态用户界面的 Qt Quick Designer*。
- 项目和编译管理 ：无论是导入现有项目还是创建一个全新项目，Qt Creator 都能生成所有必要的文件。包括对 cross-qmake 和 Cmake 的支持。
- 桌面和移动平台：Qt Creator 支持在桌面系统和移动设备中编译和运行 Qt 应用程序。通过编译设置您可以在目标平台之间快速切换。
- Qt 模拟器：Qt模拟器是诺基亚 Qt SDK的一部分，可在与目标移动设备相似的环境中对移动设备的 Qt 应用程序进行测试。



四 、Qt的历史

- 1996年Qt 上市

- Qt 已成为数以万计商业和开源应用程序的基础
- Qt 的软件授权机制具有经受市场检验的双重授权（开源与商业）模式
- Qt Software 的前身为 Trolltech（奇趣科技）。Trolltech （奇趣科技）始创于1994年
- Trolltech （奇趣科技）于2008年6月被 Nokia 收购，加速了其跨平台开发战略
- 2012年8月芬兰IT业务供应商Digia全面收购诺基亚Qt业务及其技术

五、Qt所支持的平台



1．嵌入式 Linux （Embedded Linux）

Qt for Embedded Linux® 是用于嵌入式 Linux 所支持设备的领先应用程序架构。您可以使用 Qt 创建具有独特用户体验的具备高效内存效率的设备和应用程序。Qt 可以在任何支持 Linux 的平台上运行。Qt 的直观 API，让您只须少数几行代码便可以更短的时间实现更高端的功能。

特点：1. 用于Linux 的紧凑的视窗系统；2. 用于广泛的应用程序处理器的开发；3. 移植桌面代码至嵌入式平台，或通过重新编译，反之亦然；4. 编译移除不常使用的组件与功能；5. 利用系统资源并实现本地化性能；6. 开发嵌入式设备犹如开发桌面系统一样轻松简单。

Qt 除了提供所有 工具 以及 API 与 类库 ，（ 如WebKit ） 外，Qt for Embedded Linux 还提供用于最优化嵌入式开发环境的主要组件。

- 紧凑高效的视窗系统 (QWS)：Qt 构建在标准的 API 上，应用于嵌入式 Linux 设备，并带有自己的紧凑视窗系统。基于 Qt 的应用程序直接写入Linux 帧缓冲，解除了您对 X11 视窗系统的需求。具有减少内存消耗，占位更小，可利用硬件加速图形的优势，可编译移除不常使用的组件与功能等特点。
- 虚拟帧缓冲 (QVFb)：Qt for Embedded Linux 提供一个虚拟帧缓冲器，可以采用点对点逐像素匹配物理设备显示。具有真实的测试构架，在桌面系统上嵌入式测试，模拟物理设备显示的宽度、高度与色深等特点。
- 进程间通讯 (IPC) ：IPC （进程间通讯）可以创建丰富的多应用程序用户体验。定义进程间通讯的两个主要概念即：信道与消息。可以进程并向信道发送消息，任何时候只要到一个进程便可创建信道。
- 扩展的字体格式：Qt 支持嵌入式 Linux 上的多种字体格式，包括：TrueType®, Postscript®Type1 与 Qt 预呈现字体。Qt 扩展了Unicode 支持，包括：构建时自动数据抽取和运行时自动更新。另外Qt还提供定制字体格式的插件，允许在运行时轻松添加新字体引擎。应用程序间的字体共享功能可以提高内存效率。

基本要求：

- 开发环境：Linux 内核 2.4 或更高；GCC 版本 3.3 或更高；用于 MIPS® GCC 版本 3.4.

或更高。

- 占用存储空间：存储空间取决于配置，压缩后: 1.7 - 4.1 MB，未压缩: 3.6 - 9.0 MB 。
- 硬件平台：易于载入任何支持带 C++ 编译器和帧缓冲器驱动Linux 的处理器。支持 ARM®,x86®, MIPS®, PowerPC® 。

2 · Mac 平台

Qt 包括一套集成的开发工具，可加快在 Mac 平台上的开发。在编写 Qt 时，并不需要去设想底层处理器的数字表示法、字节序或架构。要在 Apple平台上支持 Intel 硬件，Qt 客户只需重新编辑其应用程序即可。

3 · Windows平台

使用 Qt，只需一次性构建应用程序，无须重新编写源代码，便可跨多个 Windows操作系统的版本进行部署。Qt应用程序支持 WindowsVista、Server 2003、XP、NT4、Me/98 和 Windows CE。

4 · Linux/X11平台

Qt 包括一套集成的开发工具，可加快在 X11 平台上的开发。Qt 由于是 KDE 桌面环境的基础，在各个 Linux 社区人尽皆知。几乎 KDE 中的所有功能都是基于 Qt 开发的， 而且 Qt 是全球社区成员用来开发 成千上万的开源 KDE 应用程序的基础。

5 · Windows CE/Mobile

Qt 是用 C++ 开发的应用程序和用户界面框架。通过直观的 API，您可以使用 Qt 为大量的设备编写功能丰富的高性能应用程序。Qt 包括一套丰富的工具集与直观的API，意味着只须少数几行代码便可以更短的时间实现更高端的功能。

主要特点：1.硬件依存性极小；2.支持多数现有的 Windows CE 配置；3. 对于自定义的硬件配置亦轻松构建；4. 移植桌面代码至嵌入式平台，或通过重新编译，反之亦然；5. 编译移除不常使用的组件与功能；6. 利用系统资源并实现高性能；7.开发嵌入式设备尤如开发桌面系统一样轻松简单。

Qt 除了提供所有 工具 以及 API 与 类库 外，Qtfor Windows CE 还提供用于最优化嵌入式开发环境的附加功能。

- 本地化和可定制的外观：Qt 在使用时，可以支持 Windows Mobile 和 Windows CE 两种样式。 在运行时，Qt 应用程序将检测使用哪一种样式。 采用 Qt 样式表单，您只需要花费用于传统 UI 风格的少许时间和代码行，便可以轻松定制您的应用程序外观。特点：基于HTML 层叠式样式表 (CSS)；适用于全部 widget；任何熟悉 CSS 技术的人员都可以定义复杂的样式。
- 先进的文本布局引擎：Qt for Windows CE 支持 TrueType® 和点阵字体。同时 Qt 还支持扩展的 Unicode 和从右至左的书写语言。Qt 的富文本引擎增加了新的功能用于复杂的文本布局，包括制表和路径追踪，以及环绕图形的文本。

基本要求：

- 开发环境: Microsoft® Visual Studio® 2005 (Standard Edition) 或更高ActivePerl 。
- 占用存储空间：紧凑配置 - 4.8 MB，全配置 - 8.4 MB。
- 操作系统：Windows CE 5 或更高，Windows Mobile 5 或更高。
- 硬件平台：支持 ARM®, x86®，(在 SH4® 和 MIPS® 上编译) 。

6 · 塞班平台（Symbian）

Qt 通过和S60 框架的集成为 Symbian平台提供了支持。在最新版的QtSDK 1.1中我们可以直接生成可以在塞班设备上运行的sis文件。

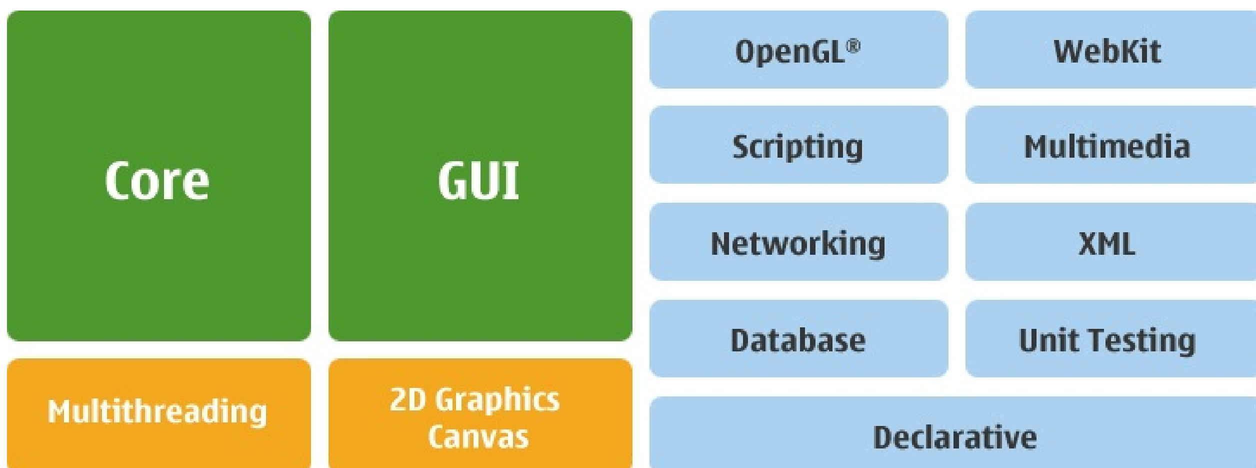
7 · MeeGo平台 (Maemo 6 现更名为 MeeGo)

Qt 是一个功能全面的应用程序和用户界面框架，用来开发Maemo 应用程序，也可跨各主要设备和桌面操作系统部署这些程序且无需重新编写源代码的。 如果您在多数情况下开发适用于Symbian、Maemo 或 MeeGo 平台的应用程序，可以使用免费 LGPL 授权方式的 Qt。

Qt将为诺基亚设备运行MeeGo (Harmattan) 提供依托，并可为所有即将推出的 MeeGo 设备中的应用程序开发提供 API，为 Qt 开发人员提供了更多平台。不久，MeeGo 设备就会完全支持 (X11) Qt 。

六、Qt类库

模块化 Qt C++ 类库提供一套丰富的应用程序生成块(block)，包含了生成高级跨平台应用程序所需的全部功能。



1 · 先进的图形用户界面（GUI）：Qt为您在桌面与嵌入式平台上开发先进的GUI应用程序，带来所有需要的功能。Qt使用所支持平台的本地化图形API，充分利用系统资源并给予应用程序本地化的界面。

- 从按钮和对话框到树形视图与表格都具有完整的控件（窗体）
- 自动缩放，字体、语言与屏幕定位识别布局引擎
- 支持抗锯齿、矢量变形以及可缩放矢量图形 (SVG)
- 具有样式API和窗体样式表，可完全自定义用户界面

- 支持嵌入式设备的硬件加速图形和多重显示功能

2 · 基于OpenGL ®与OpenGL ®Es的3D图形：OpenGL® 是一个标准的图形库，用于构建跨平台和支持硬件加速的高性能可视化应用程序。虽然OpenGL完美支持3D图形，但却不支持创建应用程序用户界面。Qt通过与OpenGL 的紧密集成解决了这一难题。

- 在您的应用程序中轻松加入3D图形
- 在嵌入式Linux 与Windows CE 平台上使用OpenGL ES和OpenGL绘画引擎
- 利用系统资源实现最佳图形性能
- 支持Windows 平台上的Direct3D®

3 · 多线程：多线程编程是一个执行资源密集型操作而不会冻结应用程序用户界面的有效典范。Qt的跨平台多线程功能简化了并行编程，另外它附加的同步功能可以更加轻松地利用多核架构。

- 管理线程、数据和对象更加轻松
- 基于Qt的信号与槽，实现跨线程类型安全的对象间通讯
- 高端API可以编译多线程程序而无须使用底端基元

4 · 嵌入式设备的紧凑视窗系统：Qt构建在标准的 API基础上，用于具有轻量级window系统的嵌入式 Linux 设备。基于 Qt的应用程序直接写入 Linux 帧缓冲，解除了您对 X11 视窗系统的需求。

- 减少内存消耗，内存占用更小
- 可以编译移除不常使用的组件与功能
- 可以利用硬件加速图形
- 在桌面系统上的虚拟帧缓冲可用于嵌入式开发与调试

5 · 对象间通讯：在开发用户图形界面中，一个常见的、重复发生系统崩溃与问题的症结根源是如何在不同组件之间进行通信。对于该问题，Qt 的解决方案是信号与槽机制，即执行Observer设计模式。我们可以简单理解为当特殊事件发生的时候，信号就被发出了，一个插槽就是一个函数，被称作特定信号的响应。

- 信号与槽机制是类型安全的(type safe)
- 任意信号都可以连接任意或多个插槽，或跨多个线程
- 简化眞正的组件编程

6 · 2D图形：Qt给您提供一个功能强大的2D图形画布，用以管理和集成大量的图形元素。

- 高精度可视化大量元素
- 将窗体互动嵌入至图形场景中
- 支持缩放、旋转、动画与变换

7 · 多媒体框架：Qt使用

Phonon多媒体框架为众多的多媒体格式提供跨桌面与嵌入式操作系统的回放功能。Phonon可以轻松将音频与视频回放功能加入到Qt应用程序当中，并且在每个目标平台上提取多媒体格式与框架。

- 以平立的方式提供多媒体内容
- 从本地文件读取媒体或读取网络上的流媒体
- 提取Mac上的 QuickTime® ，Windows 上的DirectShow® 以及 Linux 上的Gstreamer

8 · WebKit集成：Qt WebKit集成，即Qt集成了WebKit功能，WebKit是KDE项目下基于KHTML的开放源web浏览器引擎。目前 Apple®，Google™ 与Nokia等公司使用Qt WebKit集成。

- 将web与本地内容和服务整合在单一的富应用程序当中
- 快速创建整合实时web内容与服务的应用程序
- 使用集成在本地代码中的 HTML 与Java Script
- 完全控制跨平台的浏览器环境

9 · 网络连接：Qt 让您网络编程更简单，并支持跨平台网络编程。

- 完整的客户/服务器插口提取
- 支持 HTTP，FTP，DNS 与异步 HTTP 1.1
- 无论HTML 和XML或图象与媒体文件，它都可以存取所有类型的数据

10 · XML：Qt 为XML 文件以及SAX 和 DOM 协议的C++实现，提供了一个流媒体文件读写器。同时 Qt 还包含了 XQuery – 一个简单的类似 SQL的查询语言，用于解析XML文件来选择和聚合所需要的XML元素，并且将它们转换成XML输出或其它格式的输出。

- 仅需少数几行代码便可实现先进的 XML 查询
- 完全支持 XQuery 1.0 和 XPath 2.0
- 在您自己的应用程序中从XML查询、抽取和转换数据

11 · 脚本引擎：Qt 包含一个完全集成 ECMA 标准的脚本引擎。 QtScript 提供 QObject 集成，把 Qt的信号与槽机制整合成脚本，并且实现了C++ 与脚本的集成。

基于ECMA 标准的脚本语言(ECMAScript 3是JavaScript1.5的基础)

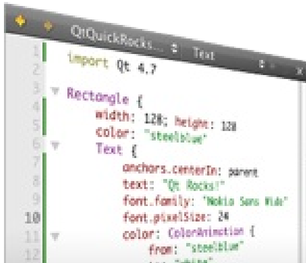
- 为简化的对象间通讯使用Qt的信号与槽机制
- 开创新的契机将脚本与您的Qt应用程序相集成

12 · 数据库：Qt 帮助您将数据库与您的Qt应用程序无缝集成。Qt支持所有主要的数据驱动，并可让您将SQL发送到数据库服务器，或者让 Qt SQL类自动生成 SQL 查询。

- 支持所有主要的数据库驱动
- 以多种视图或数据识别表单方式显示数据

七、Qt Quick介绍

Qt Quick是在Qt4.7中被引进的一项技术。Qt Quick 是一种高级用户界面技术，开发人员和设计人员可用它协同创建动画触摸式用户界面和应用程序。它由三部分构成：1.QML：像 JavaScript 一样的声明式语言；2. Qt Creator：在 Qt IDE中的直观工具；3. Qt Declarative：强大的 C++ 模块。



1．主要组成：

- QML：基于 JavaScript 的直观语言 ：QML 是一种简便易用的语言，开发人员与用户界面设计人员无需任何 C++ 知识，即可用其描绘出用户界面的外观和功能。
- 面向开发人员和设计人员的共享工具：Qt Creator IDE2.1 版将集成一套开发人员与用户界面设计人员可共享，用以创建和实施 Qt Quick 项目的通用工具。
- 通过 C++ 推动 QML 应用程序：在 Qt 库中的全新Declarative 模块支持生成动态可定制的用户界面，以及通过 C++ 拓展 QML 应用程序。

2．功能特点：

- 快速开发动画式流畅多变的用户界面：通过直观的 QML 语言和一套丰富的 QMLElements——UI 和行为生成块——您可以快速创建出令人印象深刻的用户界面，比您想象的还要快。
- 无需 C++ 知识：如果您具有 JavaScript 的经验或掌握基本的网络技术 (如 HTML 和 CSS)，您就可以通过 QML 取得非常不错的成果。
- 瞄准数以百万计的触摸屏设备：使用 Qt Quick，您可以为数以百万计的 Symbian 和 MeeGo 设备生成应用程序，或为各种类型的触摸屏消费类电子设备创建用户界面。

3．应用领域：

- 汽车信息娱乐系统 UI：Cybercom Group 的用户界面设计人员与开发人员尝试使用 Qt Quick 为其汽车信息娱乐平台设计 UI——并取得了令人满意的结果。
- 社交媒体电视：mixd.tv 使用 Qt Quick 为其跨平台网络电视应用程序创建 UI，其用户可以通过社交媒体频道访问和共享在线视频的内容。
- 联网汽车：Qt 的认证合作伙伴 Digia 很快学会了 Qt Quick 并用其创建出了包括导航、电话、游戏和音乐功能的高级汽车 UI。

八、Qt授权

Flexible Licensing GPL, LGPL, Commercial	Qt Services Support, Training, Consulting

Qt Commercial Developer License

The Qt Commercial Developer License is the correctlicense to use for the development of proprietary and/or commercial softwarewith Qt where you do not want to share any source code.

You must purchase a Qt Commercial DeveloperLicense from us or from one of our authorized resellers before you startdeveloping commercial software as you are not permitted to begin yourdevelopment with an open source licensed Qt version and convert to thecommercially license version at a later . The Qt Commercial Developer Licenseincludes a restriction that prevents the combining of code developed with theQt GNU LGPL v. 2.1 or GNU GPL v. 3.0 license versi with commerciallylicensed Qt code.

Qt GNU LGPL v. 2.1 Version

This version is available for development ofproprietary and commercial applicati in accordance with the terms andconditi of the GNU Lesser General Public License version 2.1.

Support services are available separately forpurchase.

Qt GNU GPL v. 3.0 Version

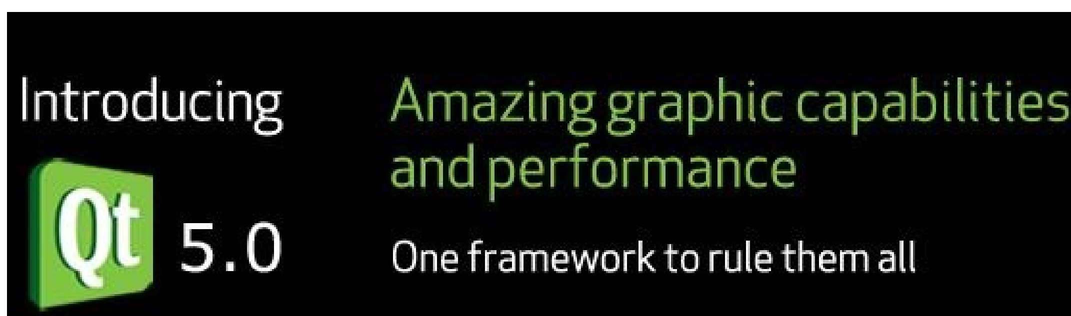
This version is freely available for the developmentof open source software governed by the GNU General Public Licenseversion 3.0 (“GPL”).

Support services are available separately forpurchase.

License Comparison Chart

	Commercial	LGPL	GPL
License cost	License fee charged	No license fee	No license fee
Must provide source code changes to Qt	No, modificati can be closed	Source code must be provided	Source code must be provided
Can create proprietary applicati	Yes - No source code must be disclosed	Yes, in accordance with the LGPL v. 2.1 terms	No, applicati are subject to the GPL and source code must be made available
Updates provided	Yes, immediate notice sent to those with a valid support and update agreement	Yes, made available	Yes, made available
Support	Yes, to those with a valid support and update agreement	Not included but available separately for purchase	Not included but available separately for purchase
Charge for Runtimes	Yes, for some embedded uses	No	No

九、Qt 5简介



Qt 5是进行Qt C++软件开发基本框架的最新版本，其中Qt Quick技术处于核心位置 。同时Qt 5能继续提供给开发人员使用原生QtC++实现精妙的用户体验和让应用程序使用OpenGl/OpenGL ES图形加速的全部功能。通过Qt 5.0提供的用户接口，开发人员能够更快的完成开发任务，针对触摸屏和平板电脑的UI转变与移植需求，也变得更加容易实现.

2012年12月19日，Digia宣布正式发行Qt 5.0。Qt 5.0是一个全新的流行于跨平台应用程序和用户界面开发框架的版本，可应用于桌面、嵌入式和移动应用程序。Qt 5 在性能、功能和易用性方面做了极大的提升，并将于明年可完全支持 Android 和 iOS 平台。Digia明确表明要使Qt 成为世界领先的跨平台开发框架。Qt 5在这个过程中具有重要的意义，它为应用程序开发

人员和产品用户提供了最好的用户体验。Qt 5极大地简化了开发过程，让他们能够更快地为多个目标系统开发具有直观用户界面的程序。它还可以很平滑的过度到新的开发模式来满足触摸屏和 Tablet 的需求。

Qt 5的主要优势包括:图形质量;中低端硬件上的高性能;跨平台移植性;支持 C + + 11; QtWebKit 2 支持的 HTML5；大幅改进QML引擎并加入新的 API；易用性并与 Qt 4 版本兼容。

- 出色的图像处理与表现能力：Qt Quick 2 提供了基于GL的工作模式，它包括一个粒子系统和一系列着色效果集合。Qt Quick 2 让复杂图形的细腻动画和变形处理变得更加容易，也确保了低端架构中2D和3D效果的平滑渲染效果和在高端架构中一样的出色。
- 更高效和灵活的研发： JavaScript和QML在保证对C++基础和Qt Widget支持上发挥着重要作用。Qt Webkit 2中一部分功能就在使用或者正考虑通过HTML 5，彻底的改变Qt
- 跨平台的移植变得更加简单：对于OS开发人员来说，由于基础模块和插件模块采用了新的架构，以及Qt跨平台性的继续强化，Qt已经能够运行在所有的环境中了。而我们的下一步计划：全面的支持iOS和Android系统，现在正在如火如荼的开发中。

Qt 通过使用 OpenGL ES，大大的增加了交付令人印象深刻的图形的能力 (OpenGL ES 是一个专门为嵌入式系统和移动设备而制定的图形应用程序编程接口)。这使它更容易开发和部署具有绚丽动画效果的 2D、3D 图形应用,这些应用在各种性能级别的嵌入式设备上得到平滑运行。例如手机、平板电脑和低成本的开发平台包括 Raspberry Pi。Qt5 新的模块化的代码库使得 Qt5 的跨平台移植性变得更简单。它包含了要点模块组和附加模块组,这种设计会减小系统代码库的尺寸。整合的 Qt 平台抽象层还强调跨平台移植性，开发人员可以通过启用开发简便性为多个目标部署。Qt 支持所有主要的桌面操作系统,包括 Windows,Mac OS X 和 Linux。嵌入式操作系统包括嵌入式 Linux、Windows 嵌入式以及最广泛部署实时操作系统的嵌入式设备——VxWorks、Neutrino 和 INTEGRITY和流行的移动操作系统等等。Qt WebKit 2 集成浏览器引擎，允许轻松集成 web 内容和应用程序。它将使 HTML5 开发人员感觉轻松自如，基于 Qt WebKit 2，能够开发出兼顾响应能力和本地代码强大功能的混合应用。这些应用可以提供大量的动态内容。

只需要一个简单的重新编译，就可以直接迁移之前为 Qt4 开发的应用程序。

结语

对于Qt 和Qt Creator有了大致的了解了，已经迫不及待想马上开始Qt的学习了吧！那么从我们的Qt Creator系列教程开始吧，让你快速进入Qt的开发行列之中！

第1篇 基础（一）Qt开发环境的搭建和hello world

导语

从这一篇我们正式开始Qt编程。本篇主要讲解Qt编程环境的搭建。为了适应大多数读者的需要，同时为了避免系统环境的不同而产生不必要的问题，这里选择使用Windows系统的Qt版本。因为在前面几十篇中我们主要讲解基本Qt控件项目的桌面编程，所以没有使用SDK进行安装，而是采用了Qt库与Qt Creator分别下载安装的方式，这样就只需要下载Qt的桌面版本的库。而SDK中默认集成了Qt Creator和Qt桌面库以及Qt移动开发的库，这个会在第40篇至第50篇进行讲解。再者，鉴于Qt一次编写代码，多次编译运行的特点，在我们教程中讲解的例子都是可以直接在其他系统环境下（比如Linux系统）直接编译运行的。

环境：Windows 7 + Qt 4.8.1+ Qt Creator 2.4.1

目 录

- 一、Qt 及 Qt Creator的下载和安装
- 二、创建hello world程序
- 三、发布程序
- 四、Qt工具介绍
- 五、附录

正文

一、Qt 及 Qt Creator的下载和安装

1．下载

(已过时）

下载Qt 4.8.1：ftp://ftp.qt-project.org/qt/source/qt-win-opensource-4.8.1-mingw.exe

下载Qt Creator2.4.1：ftp://ftp.qt-project.org/qtcreator/qt-creator-win-opensource-2.4.1.exe

最新下载地址：（已过时）

所有版本的Qt下载地址：ftp://ftp.qt-project.org/qt/source/

所有版本的Qt Creator下载地址：ftp://ftp.qt-project.org/qtcreator/

最新下载地址：http://download.qt-project.org/

其中snapshots里面包含了最新测试版本；official releases里面包含了官方发布版，即最终发布版； archive里面是Qt4.7及以前版本，Qt Creator2.5及以前版本。

更新 （2013-5-1 已过时）

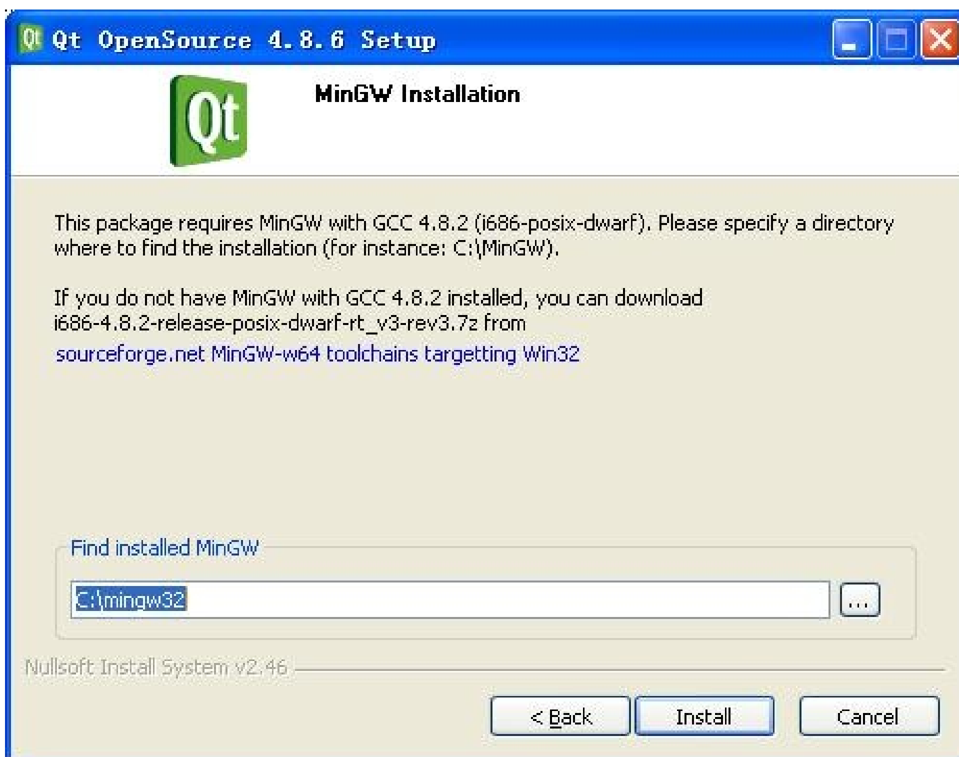
提示：在最近的Qt Creator版本（2.5.0及以后）中已经默认不再包含MinGW，需要自己手动下载安装。可以在这里下载。（注：最新的Qt 5版本中已经默认包含了Qt Creator和MinGW，需根据自己实际情况操作。

官方原文如下：

> Notefor Windows MinGW Users
>
> Wedecided to remove the custom MinGW distribution and MinGW gdb from our QtCreator-only Windows binary distribution package. The original reas toinclude it there (it was the predecessor of the Qt SDK) are since a while nowfilled by the Qt SDK. Also, updating the shipped version is a legal hassle aslong as the binaries are provided through Nokia, but we also don't want to shipstone age versi. We are working on build infrastructure for the Qt Projectitself though, that we ultimately want to use to build Qt Creator packages,snapshots, and more. Currently, on http://builds.qt-project.org , you find QtCreator snapshots for Linux and Windows, and also a Python enabled MinGW gdb(that reportedly doesn't work on Windows XP). It's still possible to installMinGW and gdb separately and register them in Qt Creator. Weare not removing the support for it from Qt Creator.
>
> Previouslyshipped MinGW: ftp://ftp.qt.nokia.com/misc/MinGW-gcc440_1.zip
>
> Previously shipped MinGW gdb: ftp://ftp.qt.nokia.com/misc/gdb/7.2/qtcreator-gdb-7.2-mingw-x86.zip
>
> Upto date MinGW: http://www.mingw.org (we might provide acompact version like the one in the old installer later)
>
> Python enabled MinGW gdb 7.4: http://builds.qt-project.org/job/gdb-windows/ (compiledon Windows 7, doesn't work on Windows XP)

更新：2014-10-1

在安装Qt 4.8.6及以后的Qt 4版本时，应该按照安装时的提示来下载相应版本的MinGW，不然编译程序无法运行。例如Qt 4.8.6安装时的提示如下图。



可以直接点击提示给的链接来下载。也可以从这里下载。

更新：（关于Qt 4.8搭配Qt Creator 2.5以后版本的MinGW和无法调试的情况，2013-7-1）

注意：Qt 5以后版本默认包含了所有需要的工具，不存在这里的情况，直接下载安装即可使用！

1．MinGW

如果是Qt 4版本，需要使用GCC 4.4，也就是MinGW需要是4.4版本的，其他新的版本均不可用。

下载：http://pan.baidu.com/share/link?shareid=1521902020&uk=2352291552

备用地址：http://builds.qt-project.org/job/mingw32-windows/lastSuccessfulBuild/artifact/mingw32-qtproject.7z

2．调试器GDB

在Qt 4.8版本，需要下载并指定GDB才能正常调试。

下载地址：http://origin.releases.qt-project.org/gdb/ 或到 社区下载页面进行下载

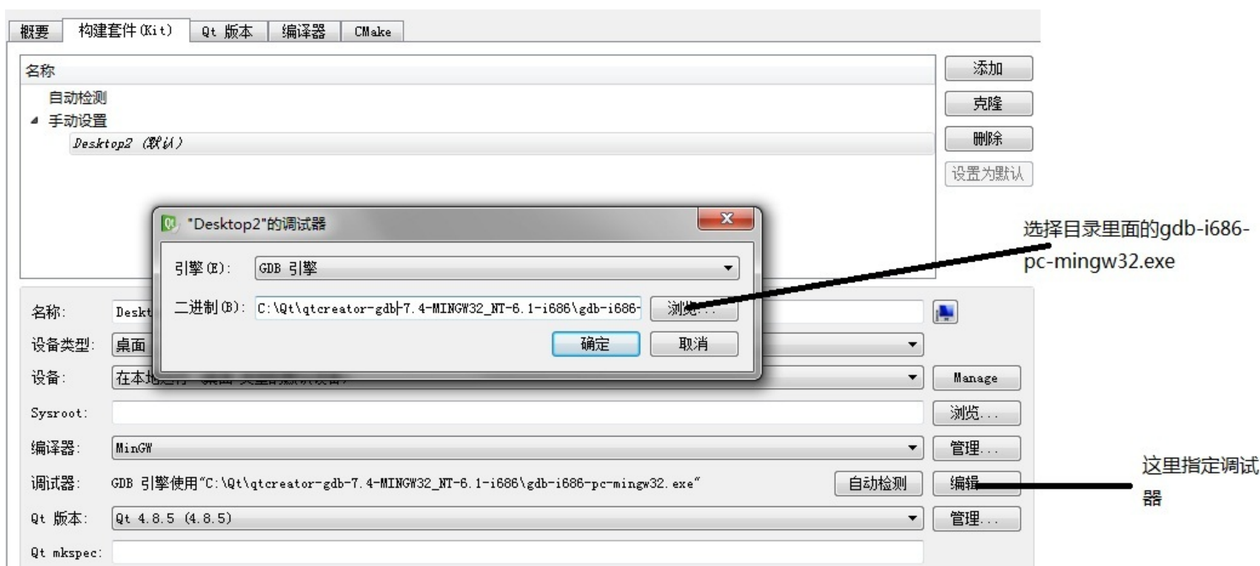
从这里面根据自己的系统来下载合适的版本。

下载完MinGW和GDB以后，将其解压到Qt的安装目录中，比如这里都解压到了C:\Qt目录中。

3．在Qt Creator中的设置。我们需要先在编译器中添加并制定gcc的路径，例如 `C:\Qt\mingw32\bin\gcc.exe` 如下图所示：

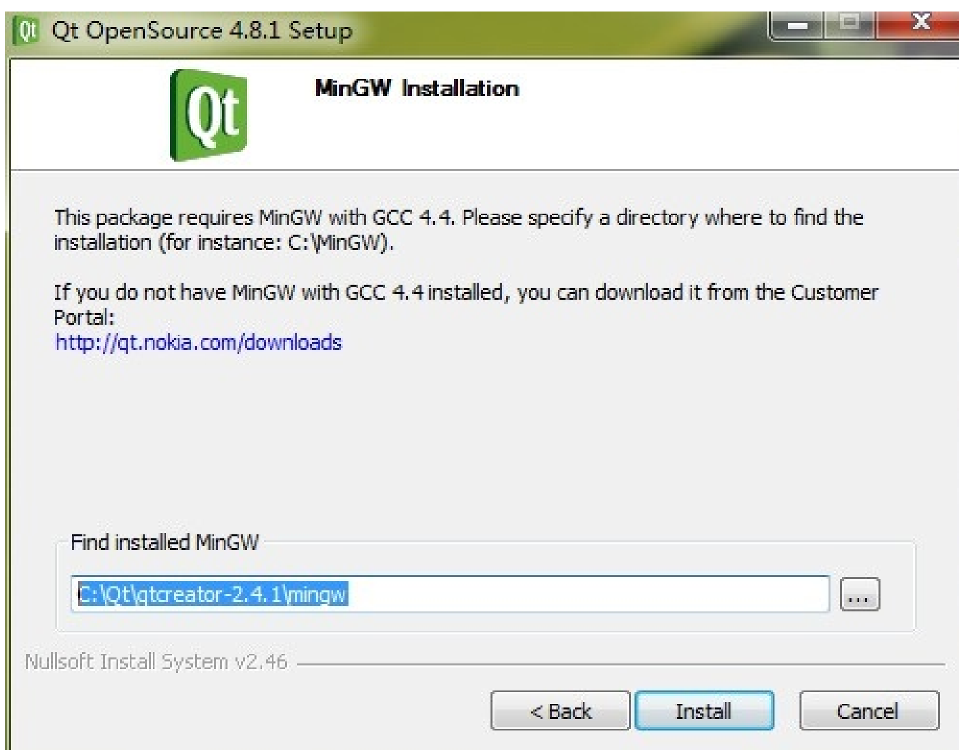
然后在Qt版本中添加并指定qmake的路径，如下图所示。

最后在构建套件中添加并指定编译器、调试器和Qt版本。如下图所示。



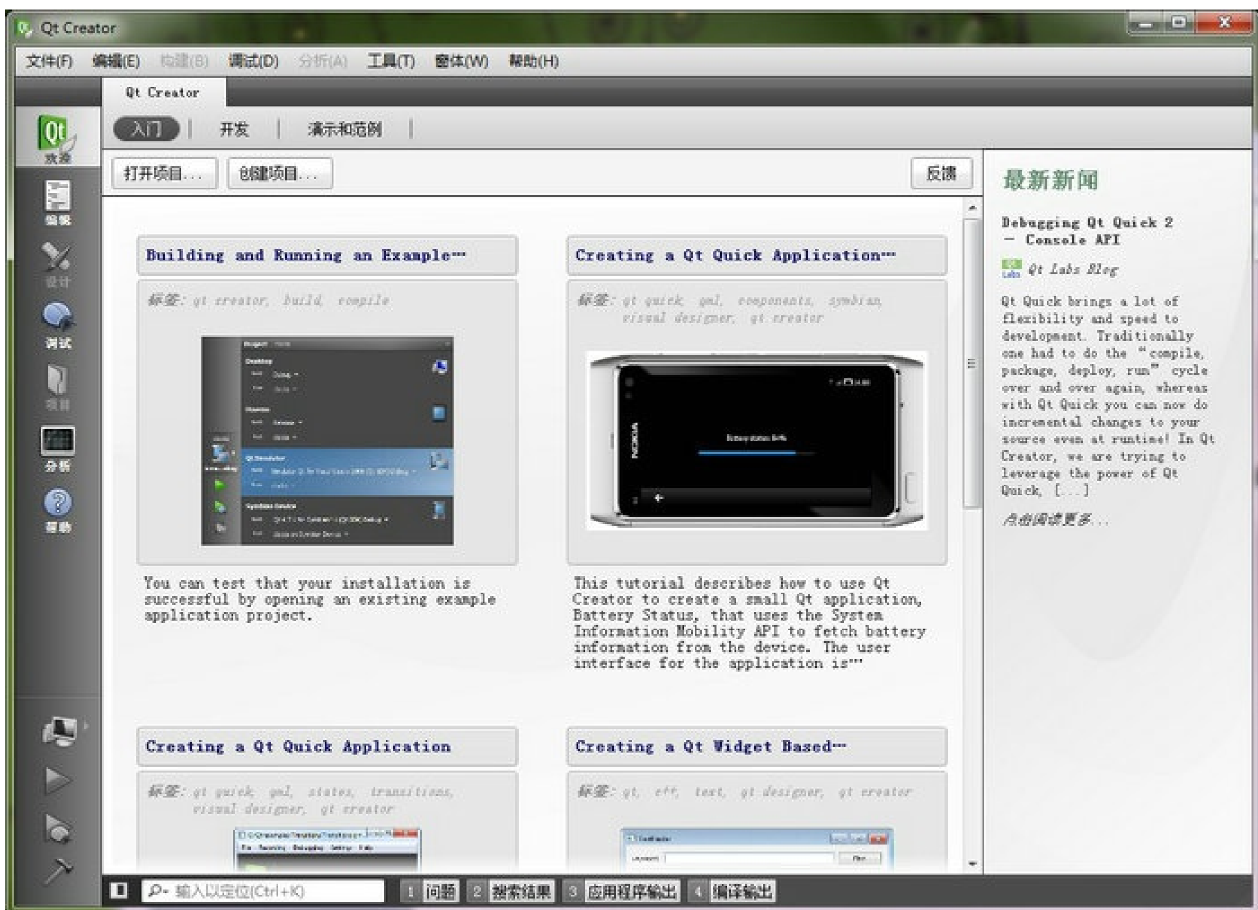
2．安装

下载完成后先安装QtCreator，采用默认选项即可，安装路径推荐使用默认的C盘，因为这样可以与教程中的一致，在以后的内容中可以避免一些不必要的问题。然后安装Qt库，当在选择mingw目录时，需要设置为前面安装的Qt Creator目录下的mingw目录。如下图所示。



二、创建hello world程序

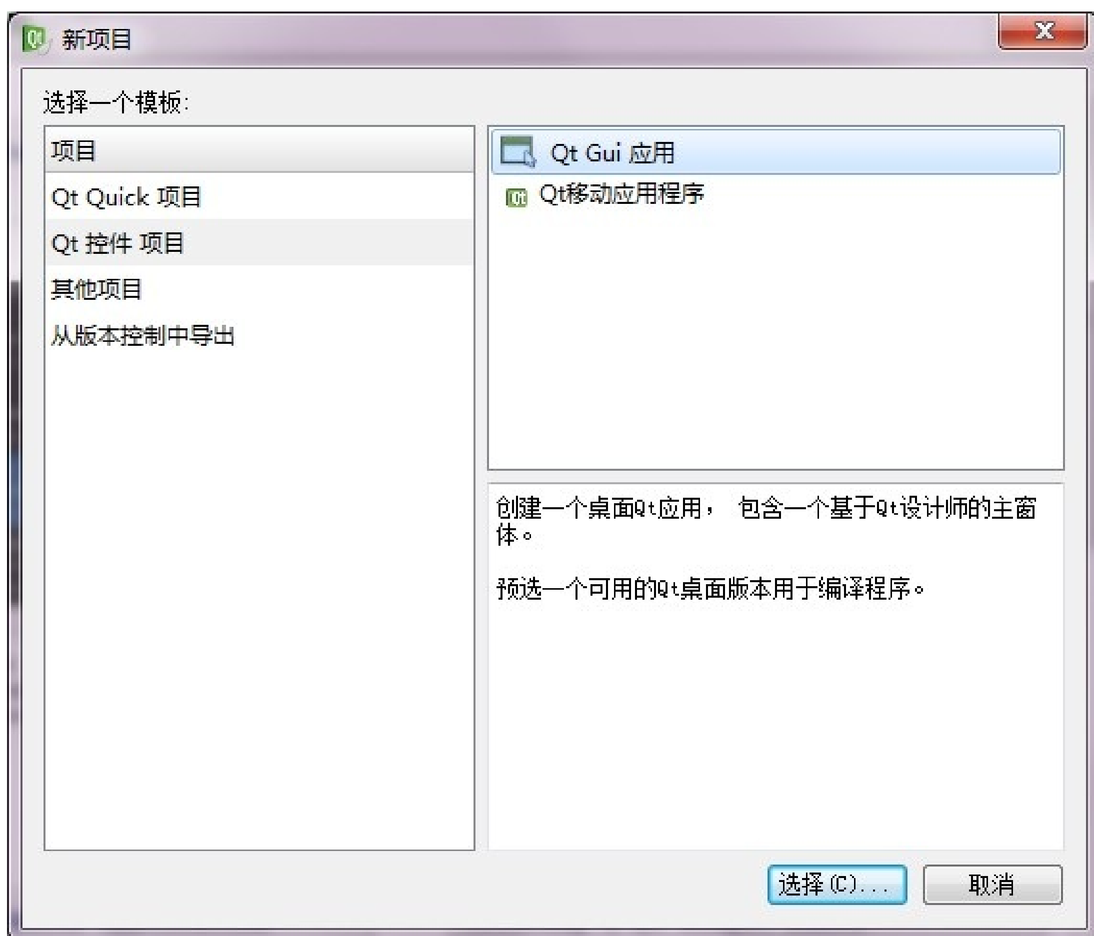
1．运行Qt Creator 从桌面上的快捷方式打开Qt Creator，进入眼帘的是Qt Creator的欢迎界面。如下图所示。



Qt Creator分为了七个模式：欢迎模式、编辑模式、设计模式、调试模式、项目模式、分析模式和帮助模式，分别由左侧的七个图标进行切换，对应的快捷键是 `Ctrl +` 数字1到7。现在显示的就是欢迎界面，这里可以看到一些入门教程、开发的项目列表、Qt提供的示例程序，也可以创建或打开一个项目。

2 · 创建项目

我们使用欢迎页面上方的“创建项目”按钮来创建新的项目（当然也可以在文件菜单中创建项目）。在项目模板中选择Qt 控件项目，然后选择QtGui应用，这样便会生成一个一般的桌面Qt图形界面项目，如下图所示。其他项目的创建会在后面的教程中讲到。



然后更改项目名称和路径，这里名称可以设置为 `helloworld` ，注意名称和路径上都不要有中文。如下图所示。

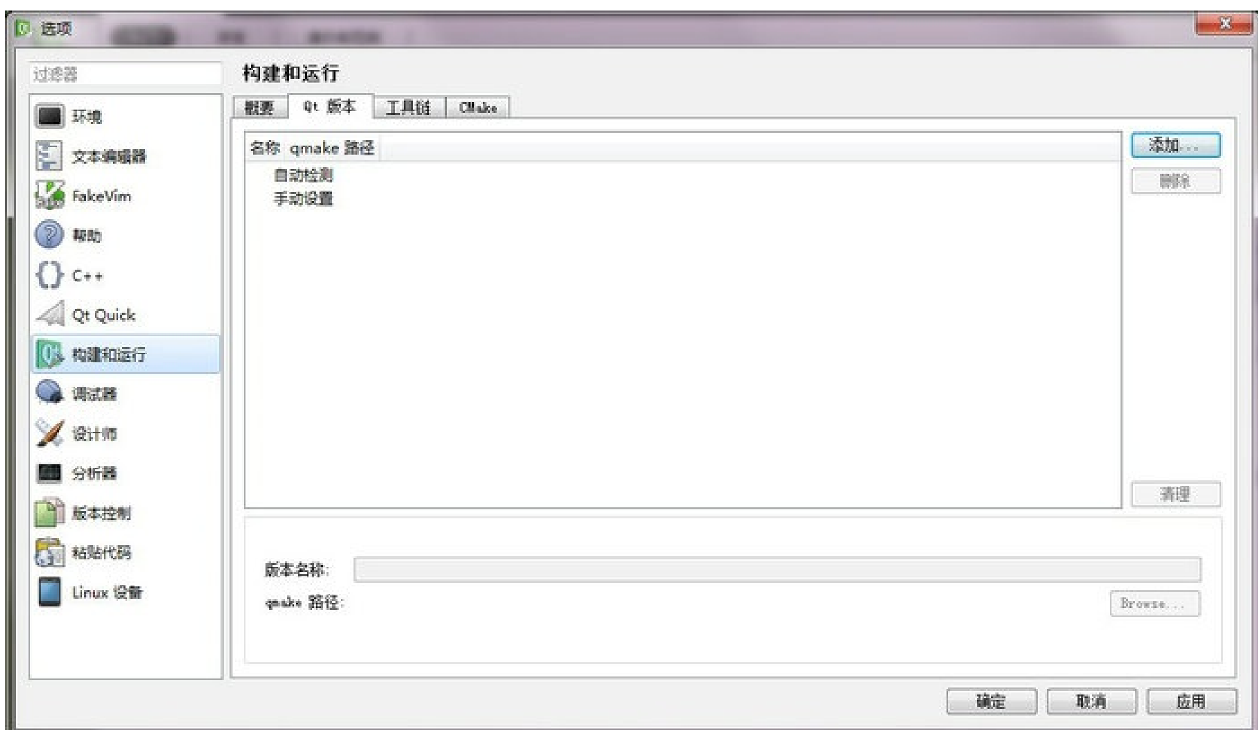


点击下一步后，会弹出目标设置对话框，这里显示没有有效的Qt版本，并提示需要在工具/选项菜单中进行设置。如下图所示。下面我们就来添加Qt版本。

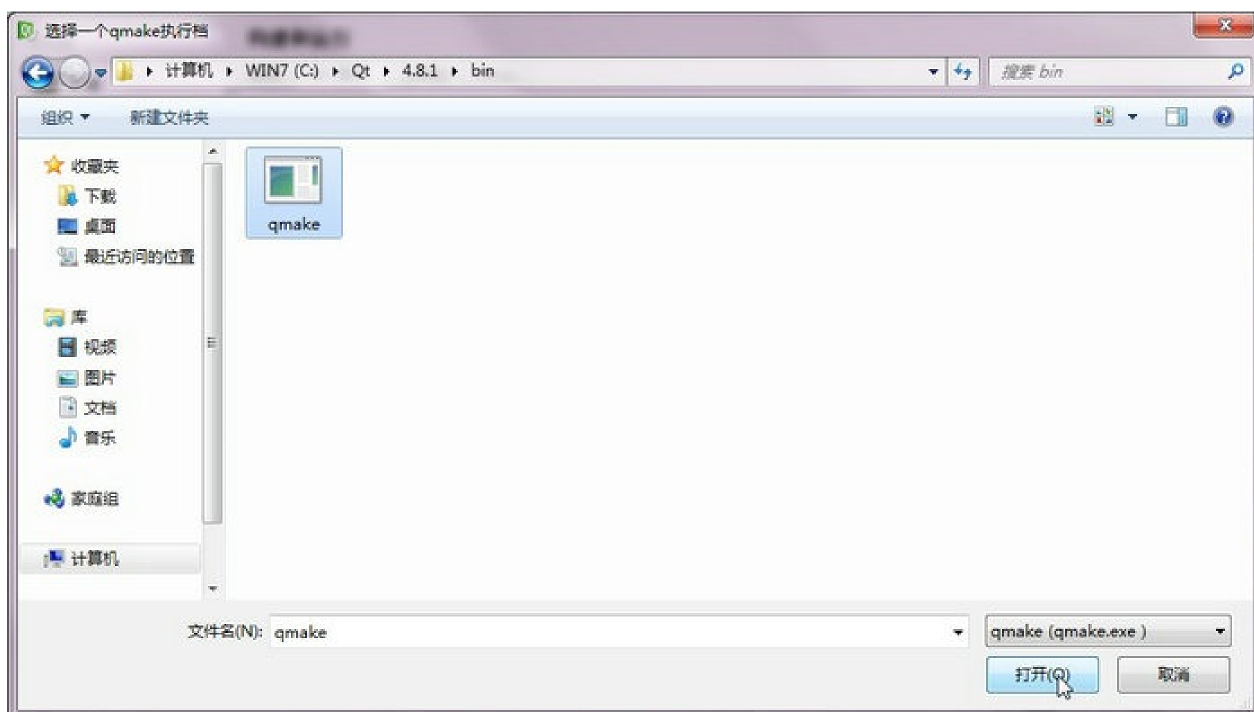


3．关联Qt库

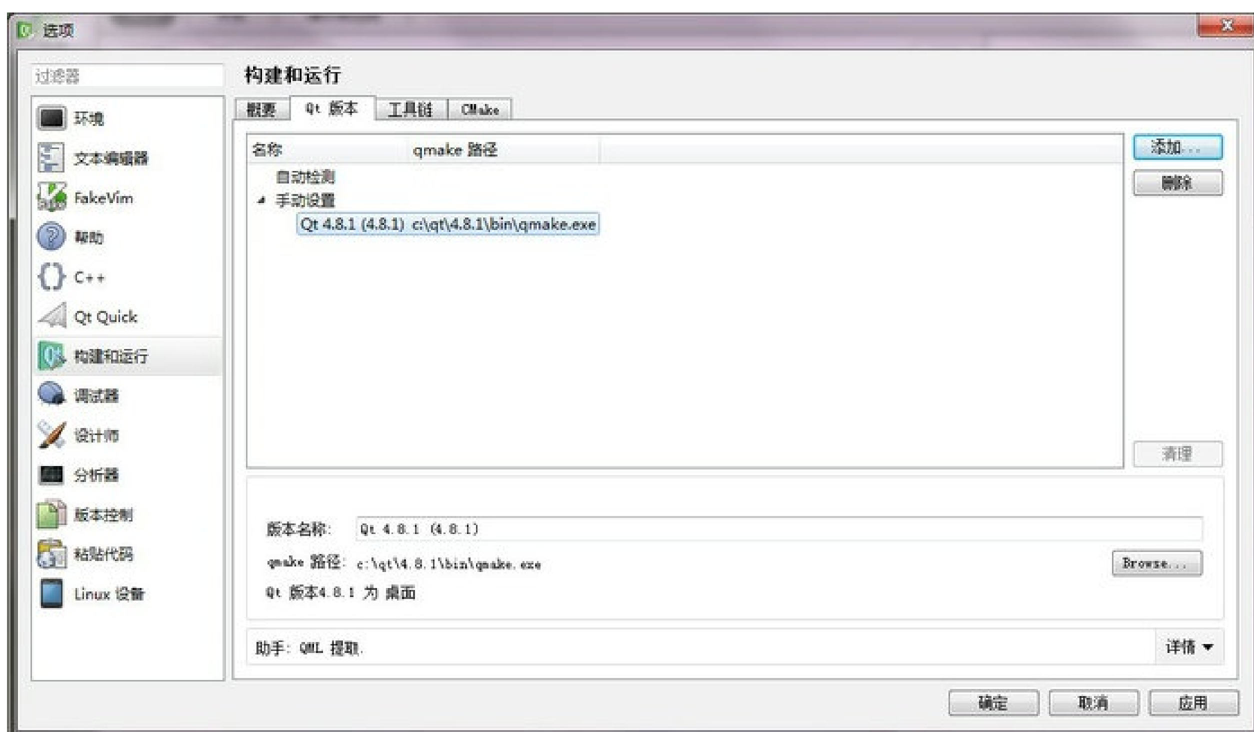
因为我们这里是分别下载Qt Creator和Qt库的方式，所以安装后它们并没有关联，这样是无法编译程序的。下面在Qt Creator中关联Qt库。打开工具→选项菜单，然后选择“构建和运行”一项，再进入Qt版本选项卡。如下图所示。



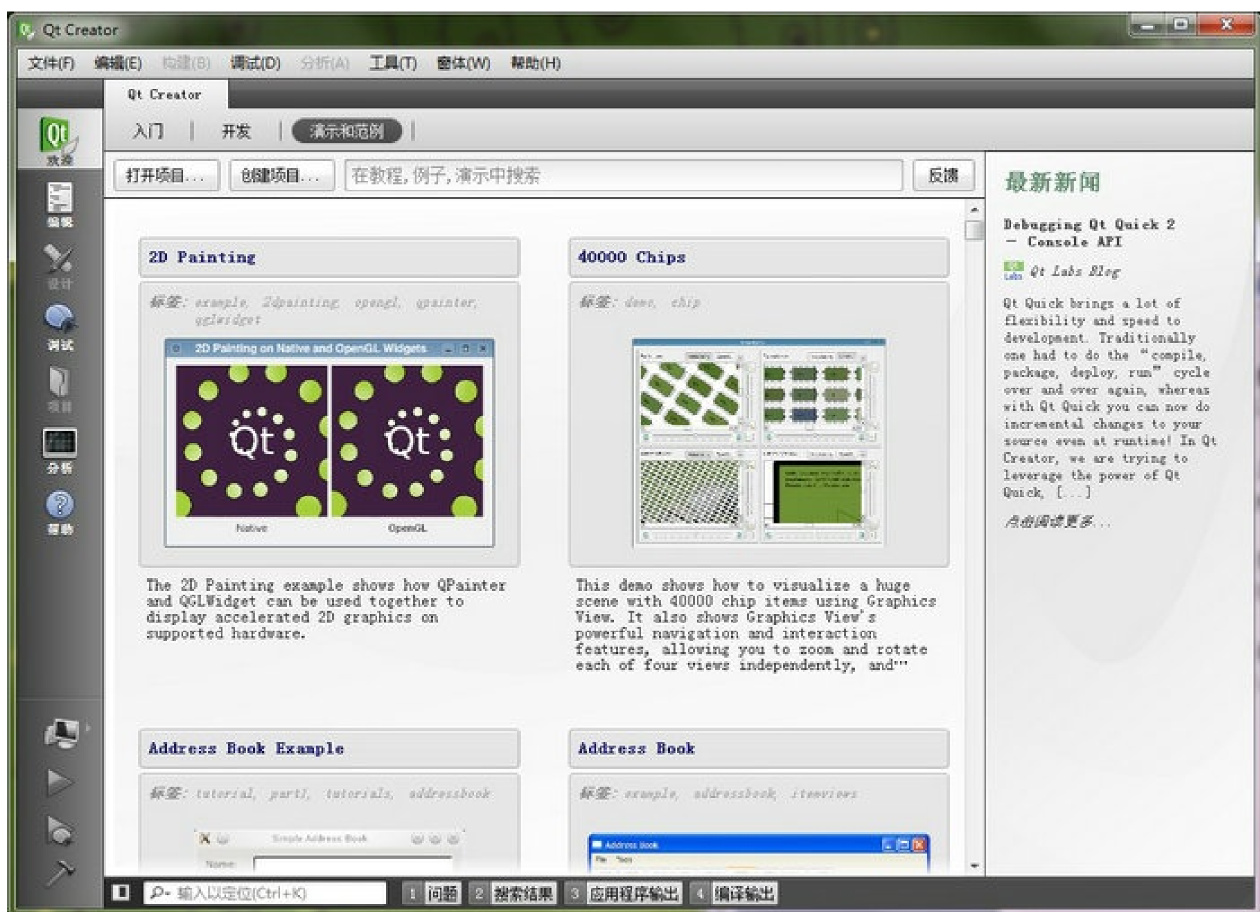
我们可以手动设置Qt版本的关联，现在点击右上角的“添加”按钮，然后会让选择 `qmake.exe` 文件，我们在Qt（不是Qt Creator）安装目录的bin目录中找到该文件并打开。如下图所示。



现在已经默认生成了版本信息，我们点击确定按钮即可。如下图所示。



当设置完Qt版本，再次回到欢迎界面后，可以发现“演示和范例”中已经显示出了各种示例程序，大家可以打开自己需要的一个例子。这个我们先不进行讲解，下面继续来完成hello world程序。

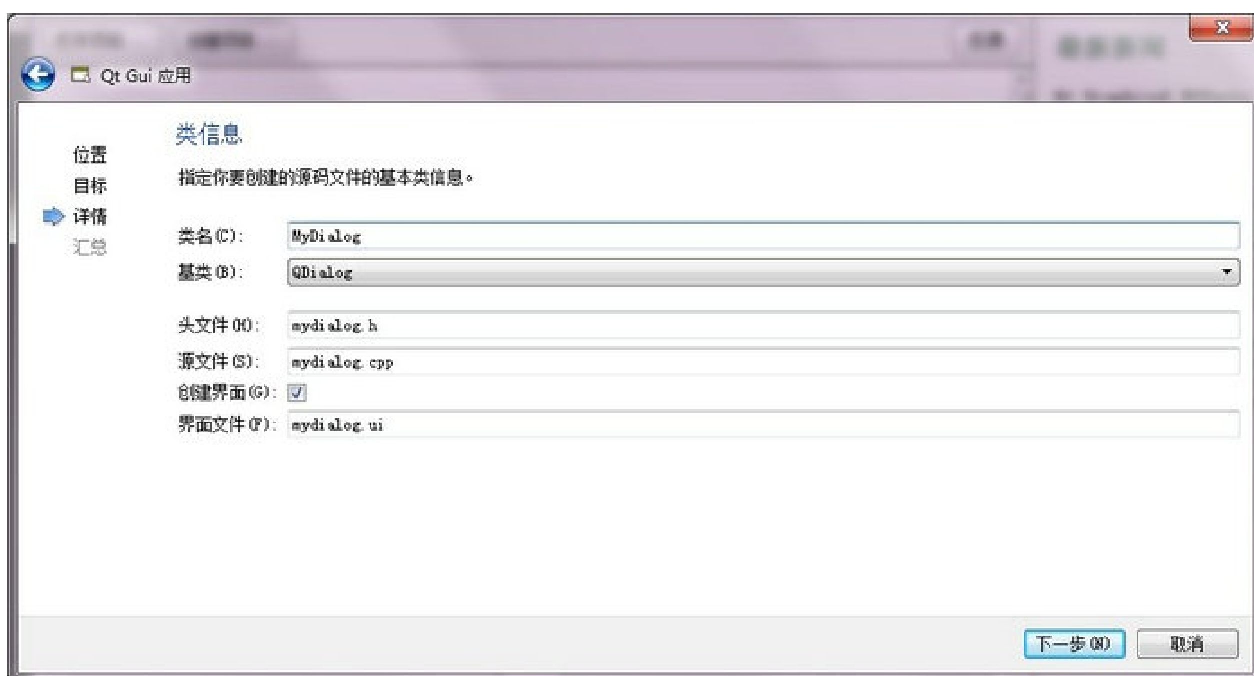


4・完成hello world项目

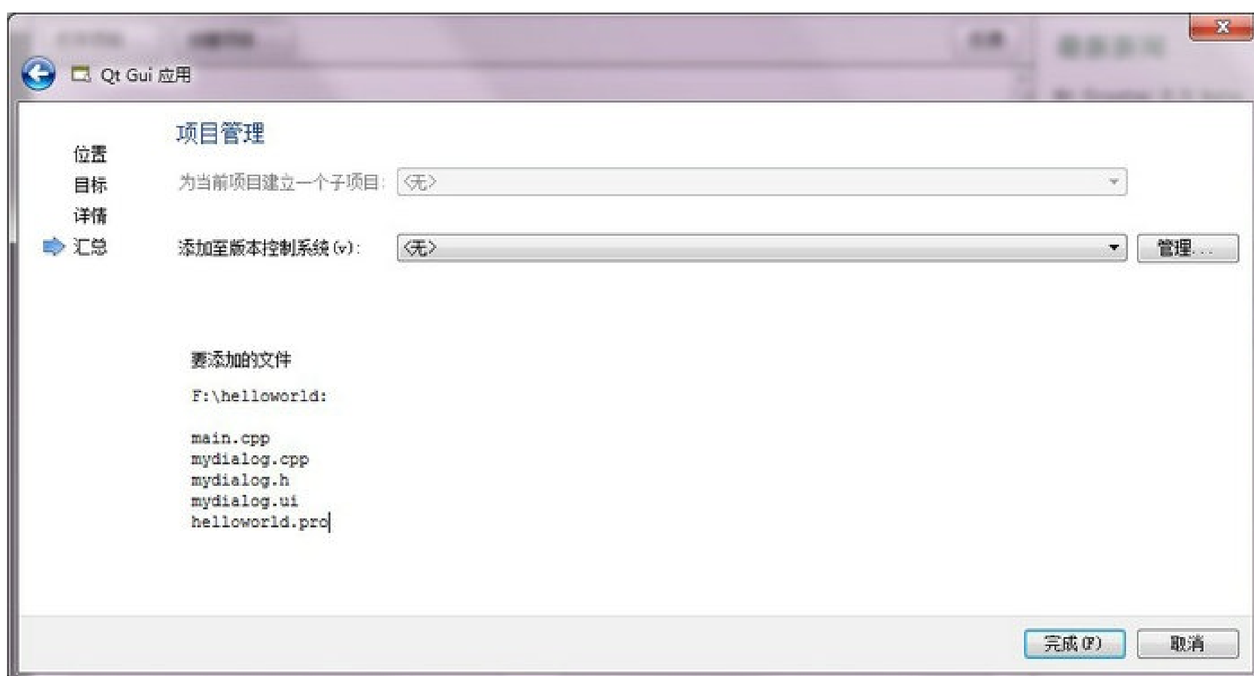
还按照前面的流程创建项目，在目标设置页面默认选择为了桌面Qt版本，因为现在我们只关联了这一个桌面版本的Qt库，所以只能编译为桌面程序。如下图所示。这里可以选中"使用影子构建"，这样编译生成的文件会和源码分别存放，这个在下面的内容中会看到。



点击下一步，在显示的类信息中将基类选择为 `QDialog` ，就是说我们将程序设置为了一个对话框，然后将类名更改为 `MyDialog` 。如下图所示。

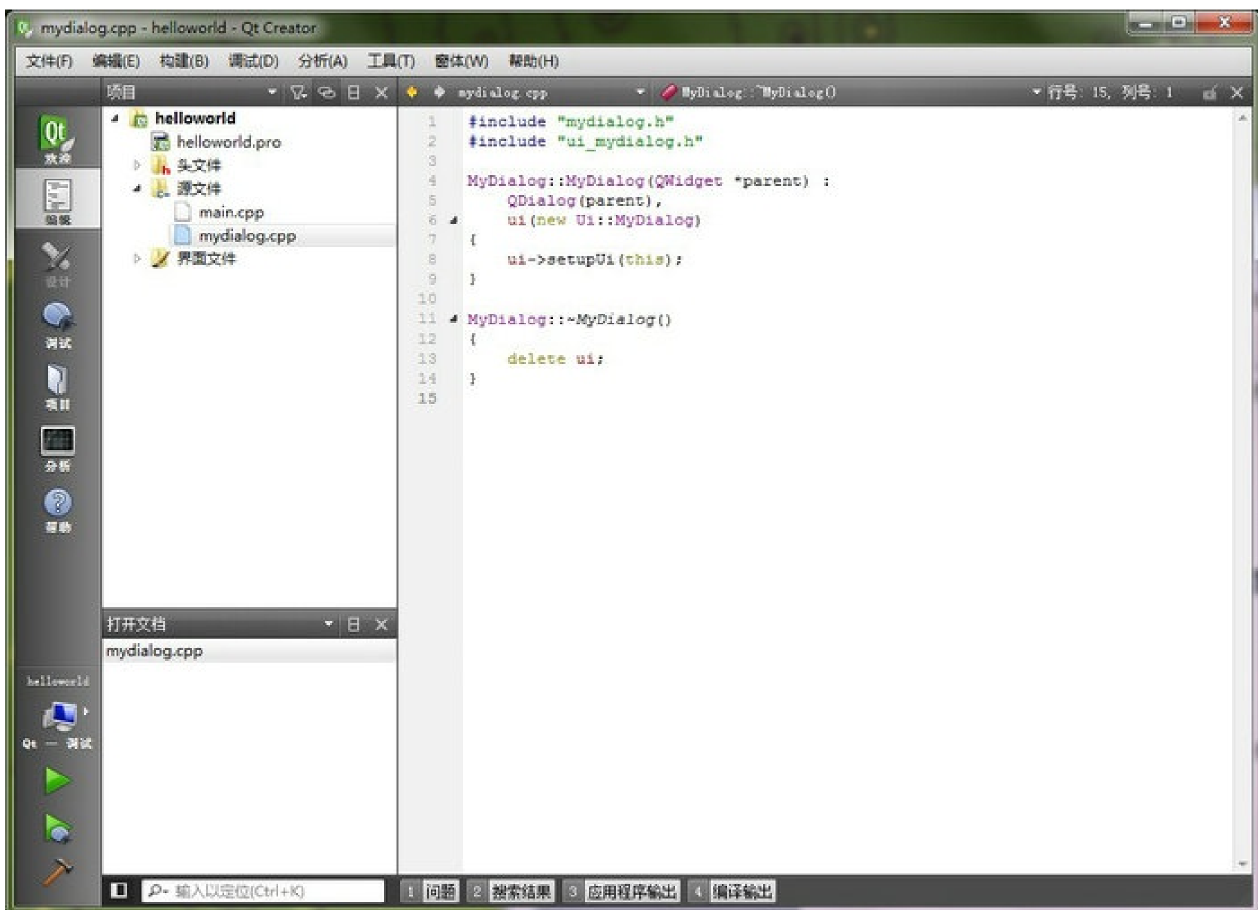


再点击下一步进入汇总页面，这里可以选择版本控制系统，我们这里没有用到，所以不进行设置，点击完成按钮来完成项目的创建。如下图所示。



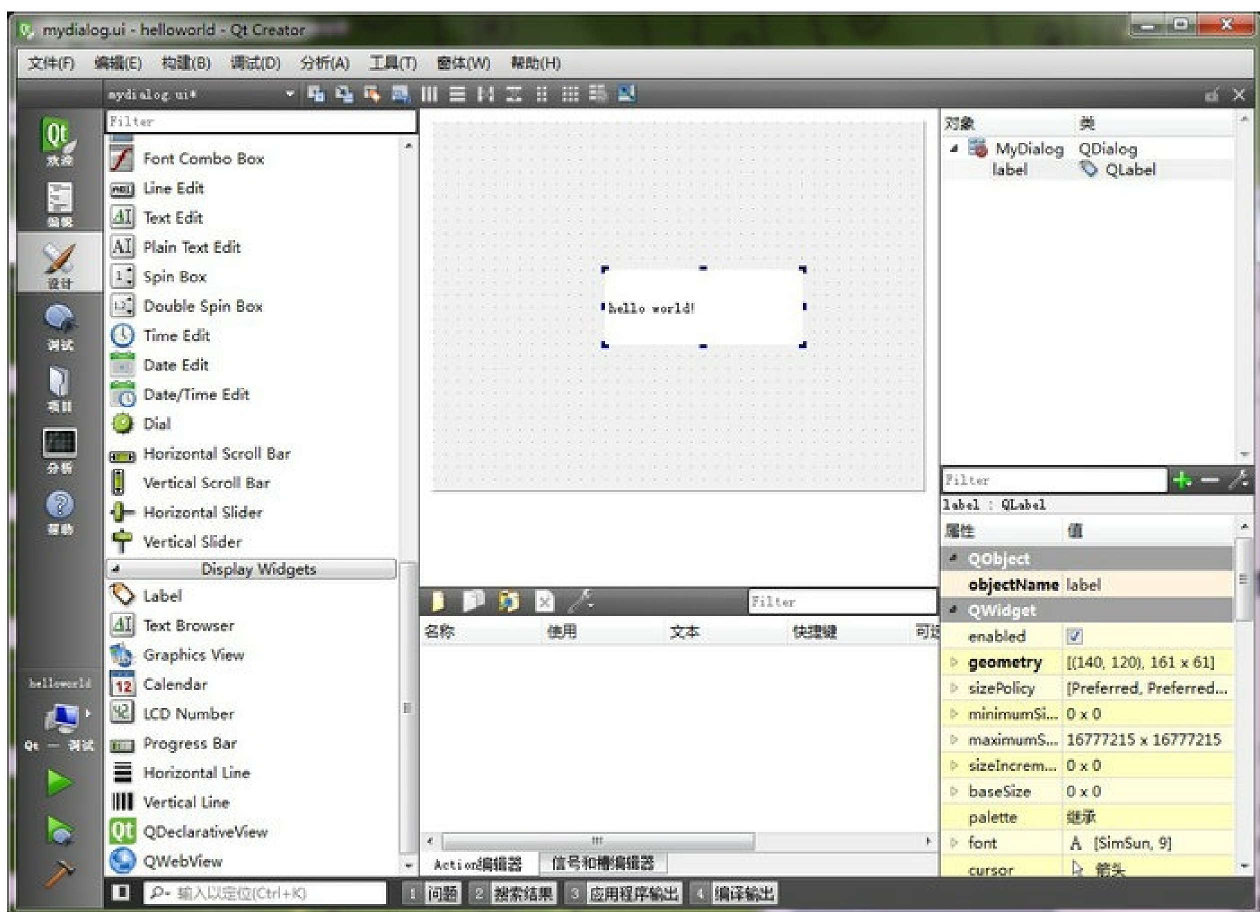
5·编辑运行项目

创建完项目后会进入编辑模式，这里可以对项目文件进行查看和编辑。左侧是项目文件的列表，这里将项目中的文件分为了头文件、源文件等，进行分类显示。除了显示项目文件，还可以通过下拉菜单来选择类视图、大纲等内容。在右侧就是代码编辑区域，这里对关键字进行了高亮显示。如下图所示。



我们双击界面文件中的 `mydialog.ui` 文件，进入设计模式。在这里可以对界面进行可视化设计，也就是所见即所得。左侧的是一些常用部件，可以直接拖动到界面上；右侧是对象和类列表，下面是部件的属性编辑窗口；在中间，上方是主设计区域，显示了窗口的主界面，下面是Action编辑器以及信号和槽编辑器窗口。

我们从左侧部件列表中找到Label标签部件并拖动到界面上，然后双击，更改其显示文本为“helloworld”，如下图所示。



下面我们单击Qt Creator左侧的 运行按钮来编译运行程序，这时会弹出保存修改对话框，如下图所示。这里选中“构建之前总是先保存文件”，然后点击保存所有文件按钮。



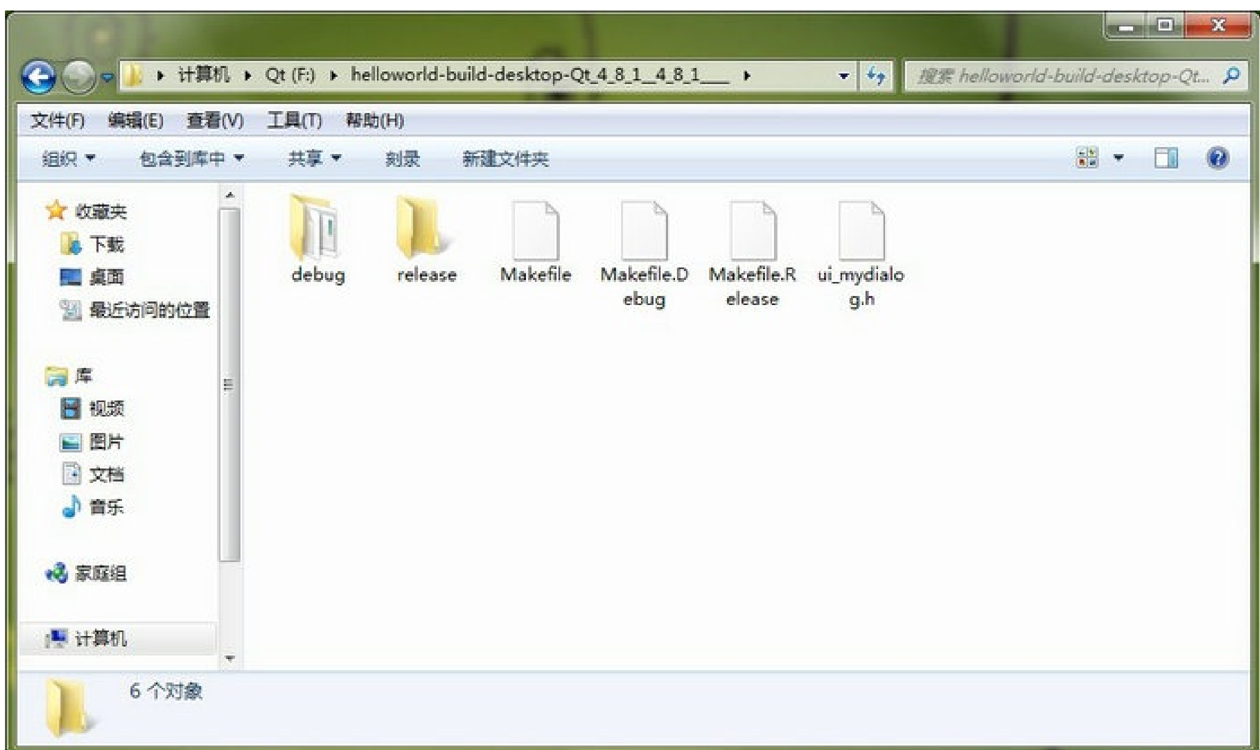
最后hello world程序成功运行，效果如下图所示。



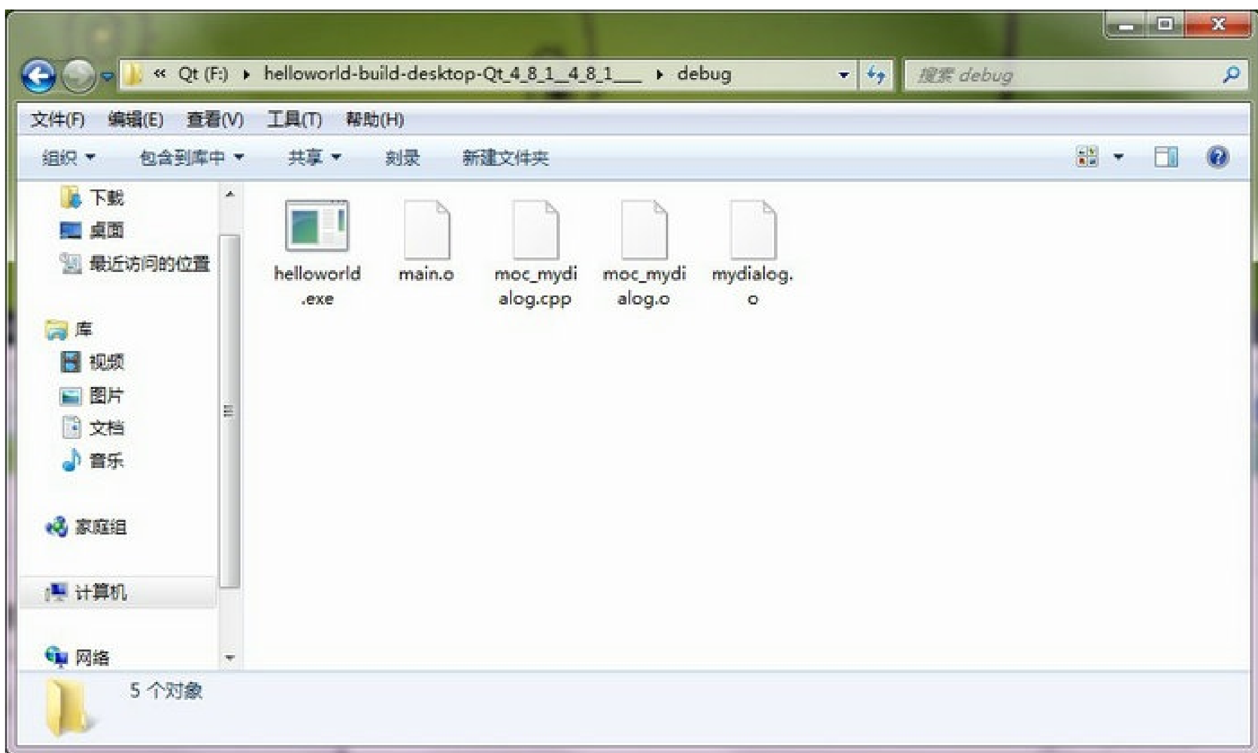
三、发布程序

1．查看工程目录

这里会发现多了一个 `helloworld-build-desktop-Qt_4_8_1__4_8_1___` 目录，里面存放的就是编译生成的文件。这就是前面创建项目讲到的“使用影子构建”，如果没有选中这个，那么生成的文件就会和源码在同一个目录里。该目录的内容如下图所示。



这里有两个目录：`debug` 和 `release`，分别用于存放debug方式和release方式编译生成的可执行文件。因为编译时默认是 `debug` 版本，所以现在 `release` 目录中是空的。打开 `debug` 目录，可以看到生成的可执行文件 `helloworld.exe` 如下图所示。



此时双击 `helloworld.exe` 文件，会弹出系统错误提示框，表明丢失了 `mingwm10.dll` 文件。如下图所示。

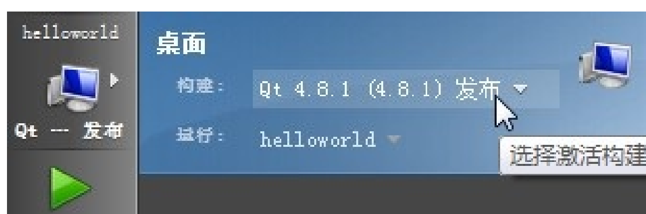


其实我们可以在Qt安装路径下找到该文件，我这里是在 `C:\Qt\4.8.1\bin` 中，将其复制到 `debug` 目录里面，然后还会提示缺少其他几个dll文件，依次将它们复制过来即可。完成后 `helloworld.exe` 就可以运行了。其实也可以先设置环境变量，以后在本机就可以直接运行生成的可执行文件了，这个可以参考下面的附录。

2．编译release版本程序

可以看到debug版本的可执行文件需要的dll文件是很大的，因为其中包含了调试信息。而我们实际发布软件是使用的release版本，下面我们就来编译release版本的helloworld程序。如果前面关闭了Qt Creator，那么需要在Qt Creator中再次打开helloworld项目，可以从欢迎模式的开发页面中打开最近使用的项目，也可以从开始菜单中打开，还可以将源码目录中的.pro文件直接拖入到QtCreator来打开。

然后将版本设置为release版本，也就是发布版本。如下图所示。完成后运行程序即可。



最后，可以从 `release` 目录中将 `helloworld.exe` 复制出来，然后将需要的几个dll文件（跟debug版本的不是完全一样哦！）也复制过来，将它们放到一个文件夹中，打包进行发布。

补充：如果要给生成的exe可执行文件更换一个自定义图标，可以这样做：

1．在项目中添加一个 `myapp.rc` (名字可以随意)文件，然后在里面输入下面一行代码：

```
IDI_ICON1               ICON    DISCARDABLE     "appico.ico"
```

这里的 `appico.ico就` 是自己的.ico图标文件；

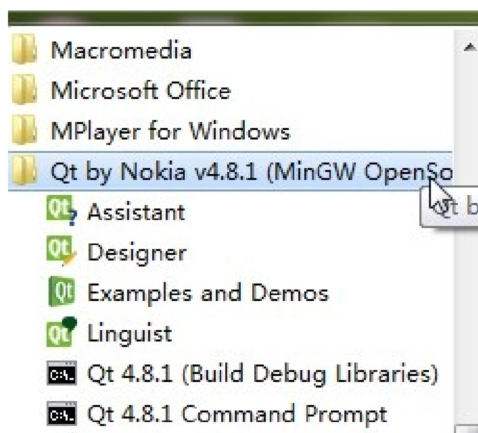
2．在 `.pro` 项目文件中添加下面一行代码：

```
RC_FILE = myapp.rc
```

3．重新编译

四、Qt工具介绍

安装好Qt后，会在开始菜单生成一个目录，如下图所示。



这里是Qt提供的几个工具软件。其中Assistant是Qt助手，它已经集成到了Qt Creator中，就是帮助模式；Designer是Qt设计师，它也集成到了QtCreator中，就是设计模式；Exampleand Demos是Qt示例程序和演示程序，其中的演示程序就是一些比较大型的程序，这个我们在欢迎模式已经看到了，不过这里可以直接运行这些程序；Linguist是Qt语言家，是用来对软件进行国际化翻译的；下面的Qt 4.8.1 Command Prompt可以用来进行命令行操作，比如使用命名来编译程序等。

五、附录

前面为了运行生成的`helloworld.exe`文件，复制了一些dll文件。其实，如果只想在本机运行程序，那么不必要每次都复制这些文件，只需要将`path`环境变量设置一下即可。我们在桌面计算机（我的电脑）图标上点击鼠标右键，选择属性，然后选择高级系统设置，在这里在高级页面选择环境变量，然后在系统变量中找到`Path`变量，双击，在变量值的最后，添加上Qt的`bin`目录的路径，我这里是`;C:\Qt\4.8.1\bin`（注意，在最前面有个英文半角的分号）。如下图所示。



这样以后就不需要再复制那些dll文件了。其实，还有一种方式也不需要dll文件，那就是静态编译，不过使用静态编译的Qt程序很大，而且不够灵活，所以这里不再讲解，有兴趣的朋友可以在网上搜索一下。

结语

这一篇中通过创建一个hello world程序，主要讲解了Qt Creator开发环境的创建以及Qt程序运行发布等内容。这一篇是最基本的知识，希望大家先看完本篇再来学习下面的内容。在《Qt Creator快速入门》一书中对开发环境以及hello world程序进行了更加详细深入的讲解，有需要的童鞋可以参考一下。

涉及到的源码下载

第2篇 基础（二）编写Qt多窗口程序

导语

程序要实现的功能是：程序开始出现一个对话框，按下按钮后便能进入主窗口，如果直接关闭这个对话框，便不能进入主窗口，整个程序也将退出。当进入主窗口后，我们按下按钮，会弹出一个对话框，无论如何关闭这个对话框，都会回到主窗口。

程序里我们先建立一个工程，设计主界面，然后再建立一个对话框类，将其加入工程中，然后在程序中调用自己新建的对话框类来实现多窗口。

在这一篇还会涉及到代码里中文字符串显示的问题。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

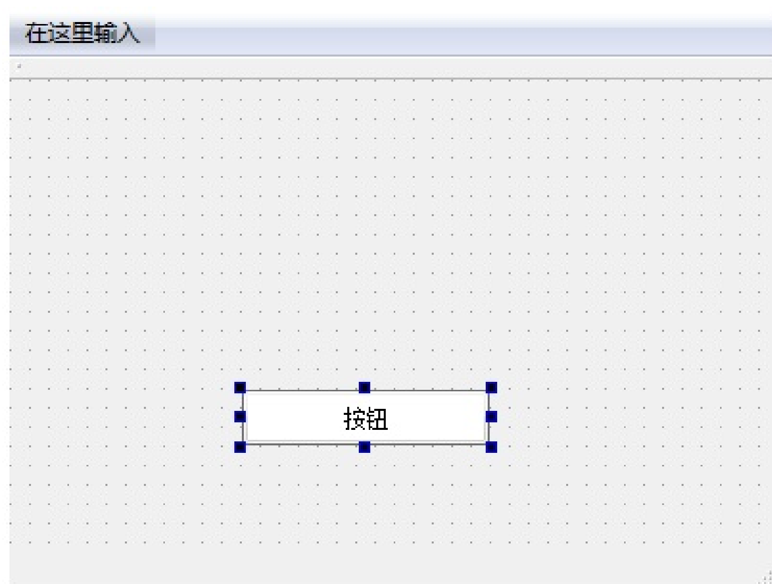
目录

正文

一、添加主窗口

1. 我们打开Qt Creator，新建Qt Gui应用，项目名称设置为 `nWindows` ，在类信息界面保持基类为 `QMainWindow` ，类名为 `MainWindow` ，这样将会生成一个主窗口界面。

2. 完成项目创建后，打开 `mainwindow.ui` 文件进入设计模式，向界面上拖入一个 `Push Button` ，然后对其双击并修改显示文本为“按钮”，如下图所示。



3．现在运行程序，发现中文可以正常显示。在设计模式可以对界面进行更改，那么使用代码也可以完成相同的功能，下面就添加代码来更改按钮的显示文本。

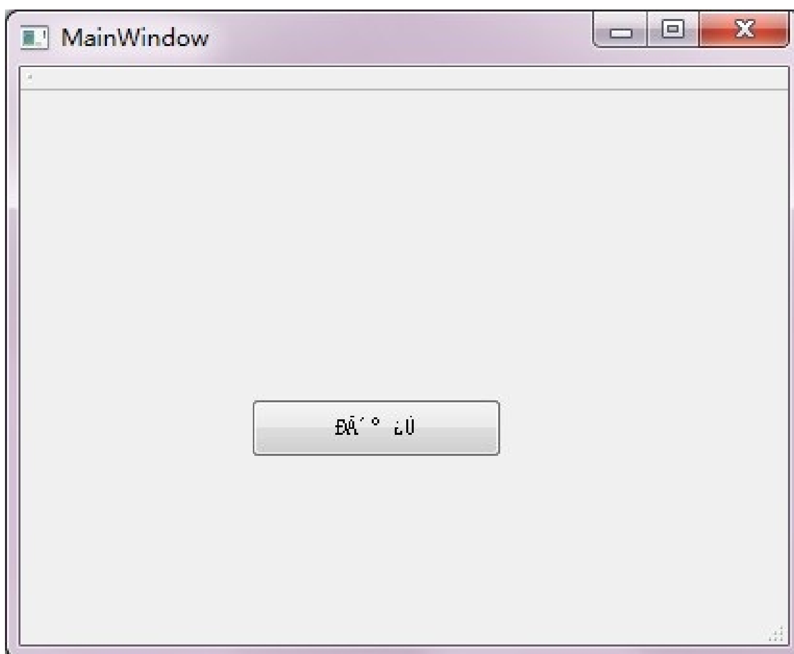
二、代码中的中文显示

1．我们点击Qt Creator左侧的“编辑”按钮进入编辑模式，然后双击 `mainwindow.cpp` 文件对其进行编辑。在构造函数 `MainWindow()` 中添加代码：

```
MainWindow::MainWindow(QWidget *parent) :
        QMainWindow(parent),
        ui(new Ui::MainWindow)
{
        ui->setupUi(this);
        ui->pushButton->setText("新窗口"); //将界面上按钮的显示文本更改为“新窗口”
}
```

这里的 `ui` 对象就是界面文件对应的类的对象，在 `mainwindow.h` 文件中对其进行了定义，我们可以通过它来访问设计模式添加到界面上的部件。前面添加的按钮部件 `Push Button` ，在其属性面板上可以看到它的 `objectName` 属性的默认值为 `pushButton` ，这里就是通过这个属性来获取部件对象的。

我们使用了 `QPushButton` 类的 `setText()` 函数来设置按钮的显示文本，现在运行程序，效果如下图所示。



2．我们发现，在代码中来设置按钮的中文文本出现了乱码。这个可以有两种方法来解决，一个就是在编写程序时使用英文，当程序完成后使用Qt语言家来翻译整个软件中的显示字符串；还有一种方法就是在代码中设置字符串编码，然后使用函数对要在界面上显示的中文字符串进行编码转换。因为翻译一个软件很麻烦，对于这些小程序，我们希望中文可以立即显示出来，所以下面来讲解第二种方法。

3．设置字符串编码，可以使用 `QTextCodec` 类的 `setCodecForTr()` 函数，一般的使用方法就是在要进行编码转换之前调用该函数，下面我们在 `main.cpp` 文件中添加代码：

```
#include <QtGui/QApplication>
#include "mainwindow.h"
#include <QTextCodec>  //添加头文件
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale()); //设置编码
    MainWindow w;
    w.show();

    return a.exec();
}
```

因为我们要在 `MainWindow` 类中进行编码转换，所以要在创建 `w` 对象以前调用该函数。这里的 `codecForLocale()` 函数返回适合本地环境的编码，当然，也可以指定编码，例如要设置为“GB2312”，可以使用下面的代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("GB2312"));
```

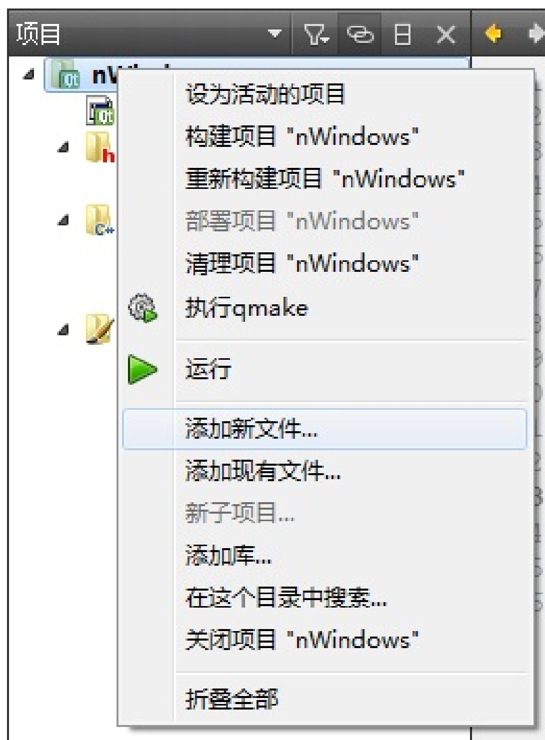
当设置完编码后，就要在显示中文字符串的地方使用 `tr()` 函数，这里我们需要将修改按钮显示文本的代码更改为：

```
ui->pushButton->setText(tr("新窗口"));
```

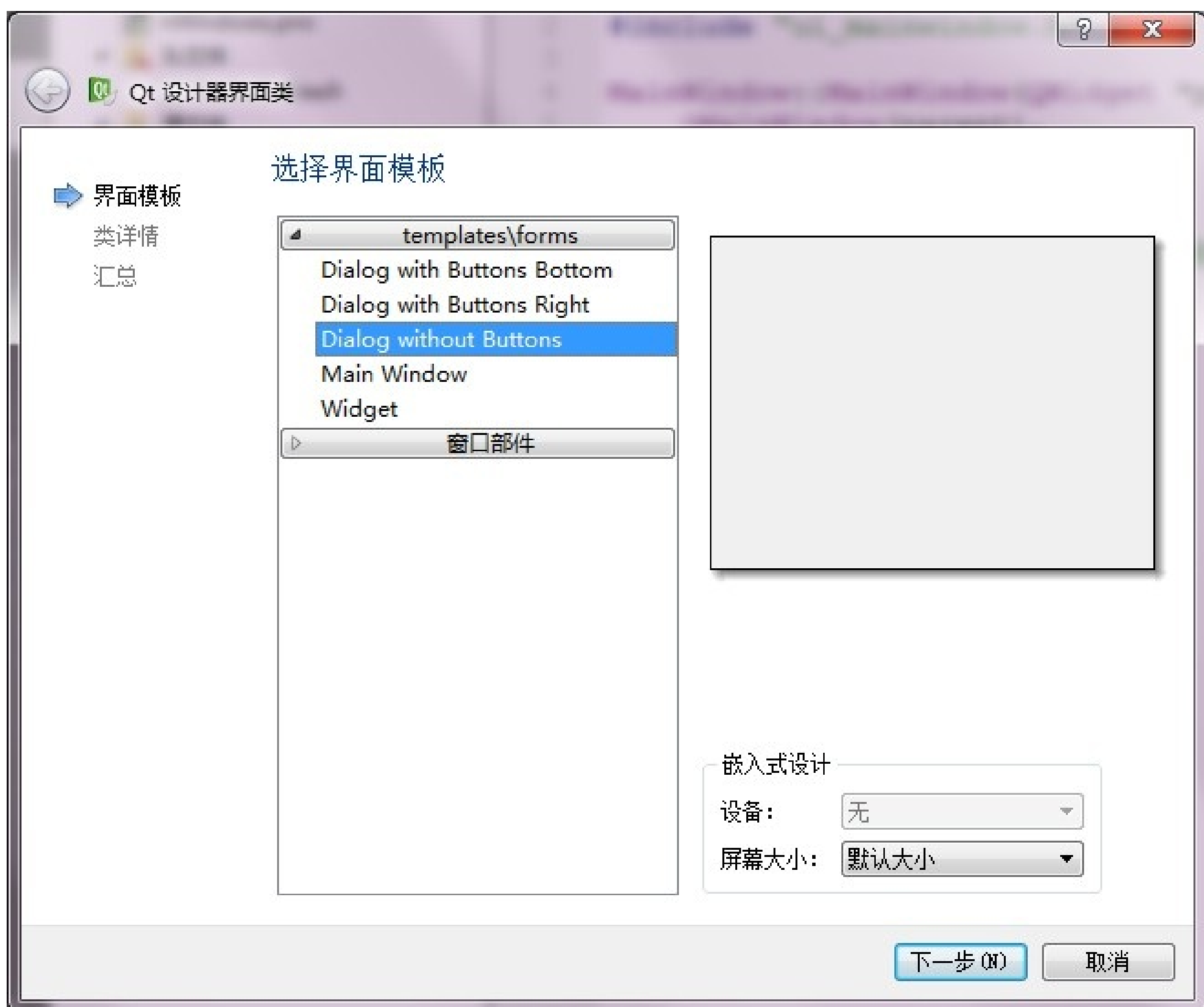
现在运行程序，可以发现中文已经可以正常显示了。这里提示一下，如果感觉编辑器中的字体太小，可以使用`Ctrl + +`（同时按下`Ctrl`和加号键）来进行放大，使用`Ctrl+ -`可以缩小。

三、添加登录对话框

1．往项目中添加新文件，这里可以在编辑模式的项目目录上点击鼠标右键，然后选择添加新文件菜单，如下图所示。当然也可以在文件菜单中进行添加。



2．模板选择Qt设计师界面类，然后界面模板选择`Dialog without Button`，如下图所示。



3·点击下一步进入类信息界面，这里将类名更改为 `LoginDlg` （注意类名首字母一般大写）。如下图所示。

Qt 设计器界面类

界面模板
类详情
汇总

选择类名

类

类名(C): LoginDlg

头文件(H): logindlg.h

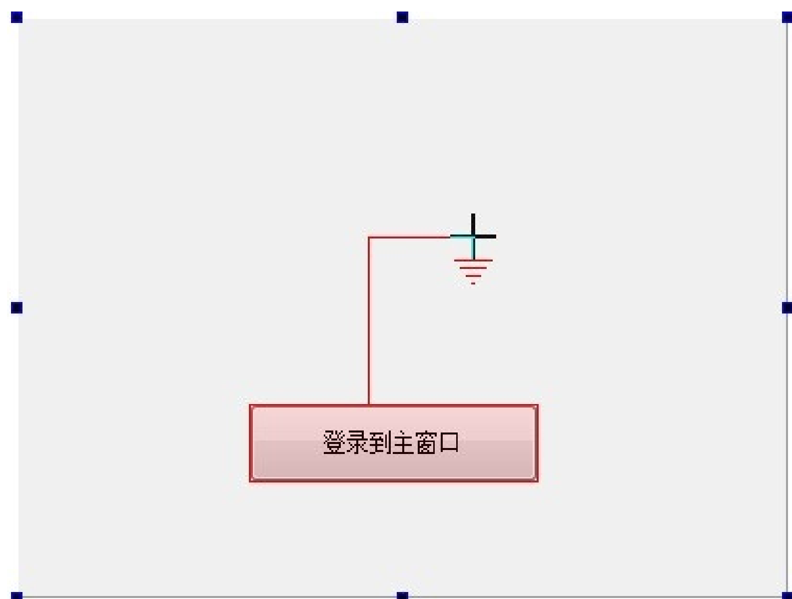
源文件(S): logindlg.cpp

界面文件(F): logindlg.ui

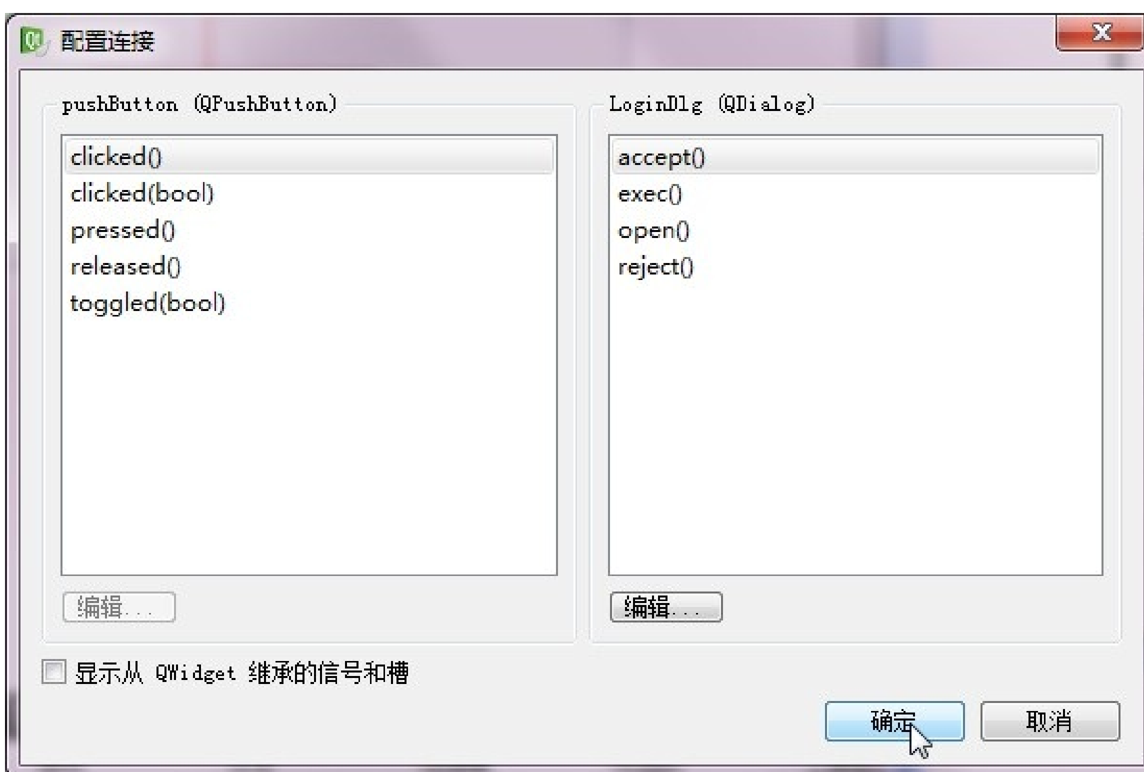
路径(P): F:\nWindows 浏览...

下一步(N) 取消

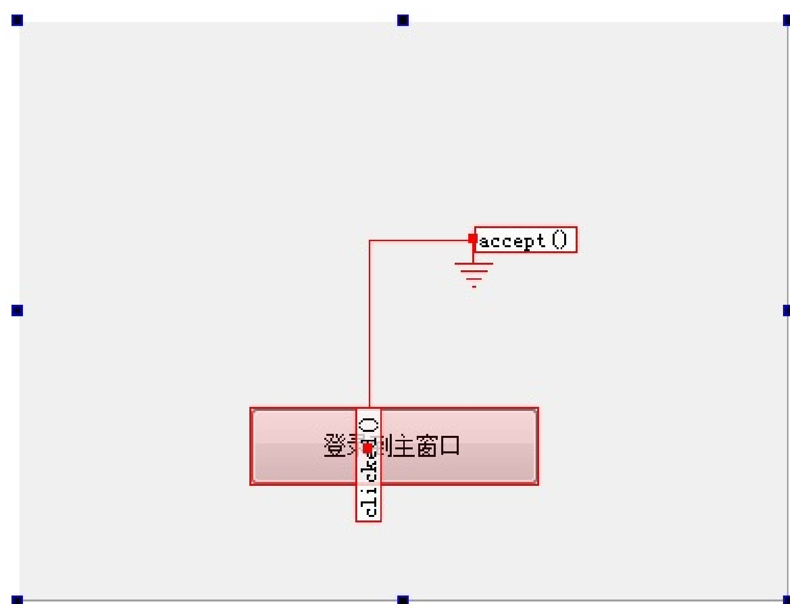
4．当完成后会自动跳转到设计模式，对新添加的对话框进行设计。我们向界面上拖入一个 `Push Button` ，然后更改显示文本为“登录到主界面”。为了实现点击这个按钮后可以关闭该对话框并显示主窗口，我们需要设置信号和槽的关联。点击设计模式上方的 图标，或者按下F4，便进入了信号和槽编辑模式。按着鼠标左键，从按钮上拖向界面，如下图所示。



当放开鼠标后，会弹出配置连接对话框，这里我们选择 `pushButton` 的 `clicked()` 信号和 `LoginDlg` 的 `accept()` 槽并按下确定按钮。如下图所示。



设置好信号和槽的关联后，界面如下图所示。



这里简单介绍一下信号和槽，大家可以把它们都看做是函数，比如这里，当单击了按钮以后就会发射单击信号，即 `clicked()` ；然后对话框接收到信号就会执行相应的操作，即执行 `accept()` 槽。一般情况下，我们只需要修改槽函数即可，不过，这里的 `accept()` 已经实现了默认的功能，它会将对话框关闭并返回 `Accepted` ，所以我们无需再做更改。下面我们就是要使用返回的 `Accepted` 来判断是否按下了登录按钮。

完成后，可以按下 ![] 或者按下F3来返回控件编辑模式。

四、使用自定义的对话框类

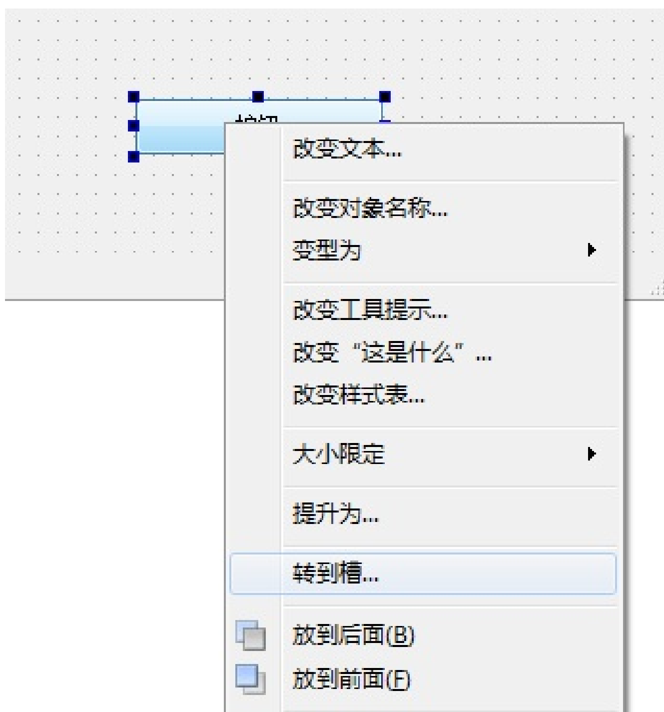
1・按下Ctrl+2返回代码编辑模式，在这里打开 `main.cpp` 文件，添加代码：

```
#include <QtGui/QApplication>
#include "mainwindow.h"
#include <QTextCodec> //添加头文件
#include "logindlg.h" //添加头文件
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    // QTextCodec::setCodecForTr(QTextCodec::codecForLocale()); //设置编码
    QTextCodec::setCodecForTr(QTextCodec::codecForName("GB2312"));
    MainWindow w;
    LoginDlg dlg;                        // 建立自己新建的类的对象dlg
    if(dlg.exec() == QDialog::Accepted) // 利用Accepted返回值判断按钮是否被按下
     {
        w.show();                      // 如果被按下，显示主窗口
        return a.exec();               // 程序一直执行，直到主窗口关闭
    }
    else return 0;             //如果没被按下，则不会进入主窗口，整个程序结束运行
}
```

在这里，我们先创建了 `LoginDlg` 类的对象 `dlg` ，然后让 `dlg` 运行，即执行 `exec()` 函数，并判断对话框的返回值，如果是按下了登录按钮，那么返回值应该是 `Accepted` ，这时就显示主窗口，并正常执行程序；如果没有按下登录按钮，那么就结束程序。

现在大家可以运行程序，测试一下效果。

2・上面讲述了一种显示对话框的情况，下面再来讲述一种情况。我们打开 `mainwindow.ui` 文件进入设计模式，然后在按钮部件上单击鼠标右键并选择转到槽菜单，如下图所示。



在弹出的转到槽对话框中选择 `clicked()` 信号并按下确定按钮。这时会跳转到编辑模式 `mainwindow.cpp` 文件的 `on_pushButton_clicked()` 函数处，这个就是自动生成的槽，它已经在 `mainwindow.h` 文件中进行了声明。我们只需要更改函数体即可。这里更改为：

```
void MainWindow::on_pushButton_clicked()
{
    QDialog *dlg = new QDialog(this);
    dlg->show();
}
```

我们创建了一个对话框对象，然后让其显示，这里的 `this` 参数表明这个对话框的父窗口是 `MainWindow` 。注意这里还需要添加 `#include <QDialog>` 头文件包含。有的童鞋可能会问，这里如果多次按下按钮，那么每次都会生成一个对话框，是否会造成内存泄露或者内存耗尽。这里简单说明一下，因为现在只是演示程序， Qt的对象树机制保证了不会造成内存泄露，而且不用写 `delete` 语句；而且因为是桌面程序，对于这样一个简单的对话框，其使用的内存可以被忽略。

当然，严谨的童鞋也可以在 `mainwindow.h` 文件中先定义一个对话框对象，并再在构造函数中进行创建，然后再到这里使用。

下面大家可以运行一下程序，查看效果。

结语

这个程序里我们实现了两类窗口打开的方式，一个是自身消失而后打开另一个窗口，一个是打开另一个窗口而自身不消失。可以看到他们实现的方法是不同的。

涉及到的源码下载

第3篇 基础（三）Qt登录对话框

导语 在前一篇的内容中已经实现了登录对话框，这里我们对其进行改进。在弹出对话框中填写用户名和密码，按下登录按钮，如果用户名和密码均正确则进入主窗口，如果有错则弹出警告对话框。

环境是：Windows 7 + Qt 4.8.1+ Qt Creator 2.4.1

目录 一、创建项目 二、登录设置

正文

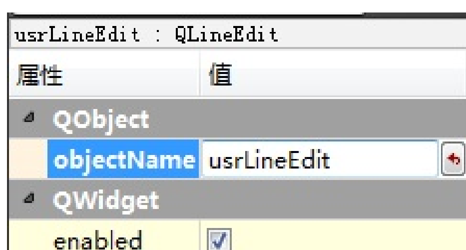
一、创建项目

1．新建Qt Gui应用，项目名称为 `login` ，类名和基类保持 `MainWindow` 和 `QMainWindow` 不变。

2．完成项目创建后，向项目中添加新的Qt设计师界面类，模板选择 `Dialog without Buttons` ，类名更改为 `LoginDialog` 。完成后向界面上添加两个标签 `Label` 、两个行编辑器 `Line Edit` 和两个按钮 `Push Button` ，设计界面如下图所示。



3．这里在属性编辑器中将用户名后面的行编辑器的 `object Name` 属性更改为 `usrLineEdit` ，密码后面的行编辑器为 `pwdLineEdit` ，登录按钮为 `loginBtn` ，退出按钮为 `exitBtn` 。如下图所示。



4．下面我们使用另外一种信号和槽的关联方法来设置退出按钮。在设计模式下面的信号和槽编辑器中，先点击左上角的绿色加号添加关联，然后选择发送者为 `exitBtn` ，信号为 `clicked()` ，接收者为 `LoginDialog` ，槽为 `close()` 。如下图所示。这样，当单击退出按钮时，就会关闭登录对话框。



5．右击登录按钮，在弹出的菜单中选择“转到槽…”，然后选择 `clicked()` 信号并确定。转到相应的槽以后，添加函数调用：

```
void LoginDialog::on_loginBtn_clicked()
{
    accept();
}
```

6．下面到 `main.cpp` 文件，更改内容如下：

```
#include <QtGui/QApplication>
#include "mainwindow.h"
#include "logindialog.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    LoginDialog dlg;
    if (dlg.exec() == QDialog::Accepted)
    {
       w.show();
       return a.exec();
    }
    else return 0;
}
```

7．这时运行程序，按下退出按钮会退出程序，按下登录按钮会关闭登录对话框，并显示主窗口。

二、登录设置

1．下面添加代码来实现使用用户名和密码登录，这里我们只是简单的将用户名和密码设置为了固定的字符串。到 `logindialog.cpp` 文件中将登录按钮的单击信号对应的槽的代码更改为：

```
void LoginDialog::on_loginBtn_clicked()
{
// 判断用户名和密码是否正确，
// 如果错误则弹出警告对话框
    if(ui->usrLineEdit->text() == tr("yafeilinux") &&
           ui->pwdLineEdit->text() == tr("123456"))
    {
       accept();
    } else {
       QMessageBox::warning(this, tr("Waring"),
                            tr("user name or password error!"),
                            QMessageBox::Yes);
    }
}
```

Qt中的 `QMessageBox` 类提供了多种常用的对话框类型，比如这里的警告对话框，还有提示对话框，问题对话框等。这里使用了静态函数来设置了一个警告对话框，这种方式很方便。其中的参数依次是： `this` 表明父窗口是登录对话框；然后是窗口标题；然后是显示的内容；最后一个参数是显示的按钮，这里使用了一个Yes按钮。大家注意还要添加该类的头文件包含，即： `#include <QMessageBox>` 。

2．下面运行程序，如果输入用户名为 `yafeilinux` ，密码为 `123456` ，那么可以登录，如果输入其他的字符，则会弹出警告对话框，如下图所示。



3．对于输入的密码，我们常见的是显示成小黑点的样式。下面点击 `logindialog.ui` 文件进入设计模式，然后选中界面上的密码行编辑器，在属性编辑器中将 `echoMode` 属性选择为 `Password` 。这时再次运行程序，可以看到密码显示已经改变了。如下图所示。



当然，除了在属性编辑器中进行更改，也可以在 `loginDialog` 类的构造函数中使用 `setEchoMode(QLineEdit::Password)` 函数来设置。

4．在行编辑器的属性栏中还可以设置占位符，就是没有输入信息前的一些提示语句。例如将密码行编辑器的 `placeholderText` 属性更改为“请输入密码”，将用户名行编辑器的更改为“请输入用户名”，运行效果如下图所示。



5．对于行编辑器，还有一个问题就是，比如我们输入用户名，在前面添加了一个空格，这样也可以保证输入是正确的，这个可以使用 `QString` 类的 `trimmed()` 函数来实现，它可以去除字符串前后的空白字符。下面将 `logindialog.cpp` 文件中登录按钮单击信号槽函数中的判断代码更改为：

```
if(ui->usrLineEdit->text().trimmed() == tr("yafeilinux")
            && ui->pwdLineEdit->text() == tr("123456"))
```

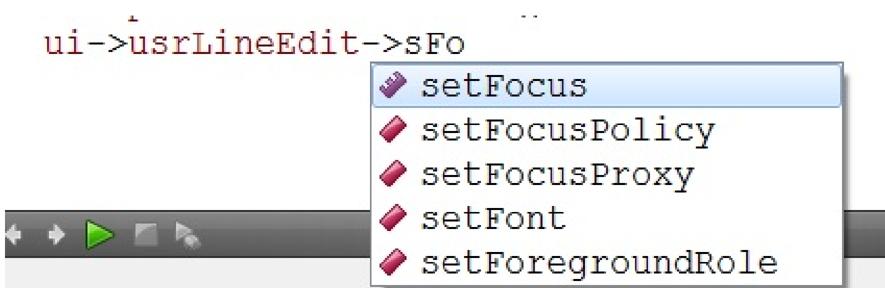
这时运行程序，已经实现相应的功能了。

6. 最后，当登录失败后，我们希望可以清空用户名和密码信息，并将光标定位到用户名输入框中。这个可以通过在判断用户名和密码错误后添加相应的代码来实现：

```
void LoginDialog::on_loginBtn_clicked()
{
    // 判断用户名和密码是否正确，如果错误则弹出警告对话框
    if(ui->usrLineEdit->text().trimmed() == tr("yafeilinux")
           && ui->pwdLineEdit->text() == tr("123456"))
    {
       accept();
    } else {
       QMessageBox::warning(this, tr("Waring"),
                            tr("user name or password error!"),
                            QMessageBox::Yes);
       // 清空内容并定位光标
       ui->usrLineEdit->clear();
       ui->pwdLineEdit->clear();
       ui->usrLineEdit->setFocus();
    }
}
```

下面运行程序，大家可以测试一下效果。

7. 这里再补充一个技巧，也就是Qt Creator的代码补全功能。Qt Creator有很强大的代码补全功能，比如输入一个关键字时，只要输入前几个字母，就会弹出相关的关键字的选择列表；输入完一个对象，然后输入点以后，就会弹出该对象所有可用的变量和函数。这里要说的是，当输入一个比较长得函数或变量名时，可以通过其中的几个字母来定位。比如说，要输入前面讲到的 `setFocus()` 函数，那么只需输入首字母 `s` 和后面的大写字母 `F` 即可，这样可以大大缩减提示列表，如果还没有定位到，那么可以输入 `F` 后面的字母。如下图所示。



我们还可以使用 `ctrl + 空格键` 来强制代码补全，不过这个一般会和我们的输入法的快捷键冲突，大家可以更改输入法的快捷键，也可以在Qt Creator中的工具→选项→环境→键盘中来设置快捷键。

结语

这一节又讲解了一种信号和槽的关联方法，还讲解了一些部件的属性设置等内容。在《Qt Creator快速入门》一书中还讲解了大量常用的部件的使用说明，大家可以参考一下。

涉及到的源码下载

第4篇 基础（四）添加菜单图标——使用Qt资源文件

导语

后面几篇里我们将介绍常用的Qt主窗口部件`QMainWindow`，主窗口部件就是一般的应用程序主窗口，它包含了菜单栏、工具栏、中心部件、状态栏和可停靠部件等。这一篇将着重介绍菜单的实现以及使用资源文件来添加菜单图标。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

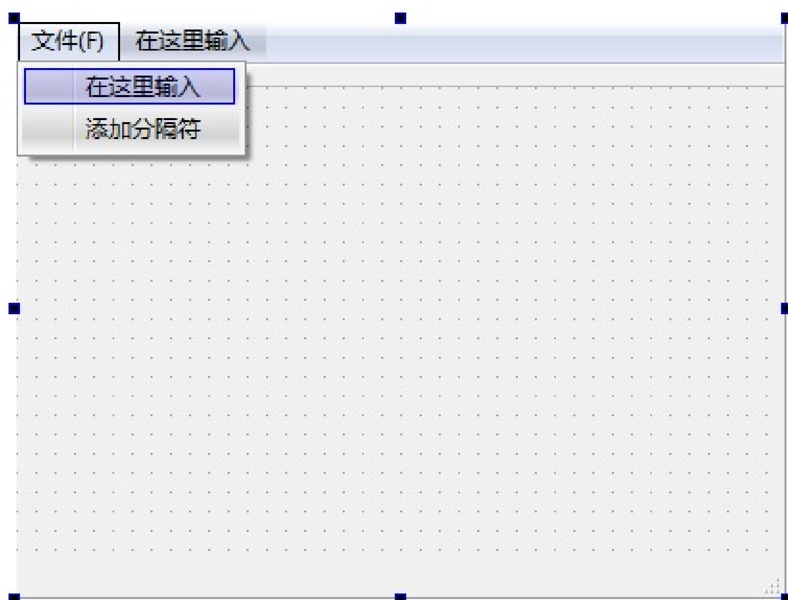
目录

正文

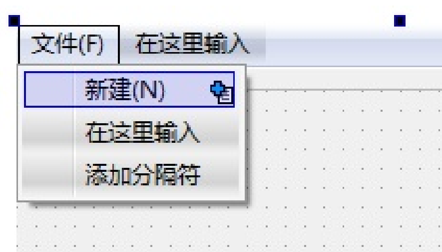
一、添加主窗口菜单

1．新建Qt Gui应用，项目名称为`myMainWindow`，基类选择`QMainWindow`，类名为`MainWindow`。

2．创建完项目后，打开`mainwindow.ui`文件进入设计模式。在这里可以看到界面左上角的“在这里输入”，我们可以在这里添加菜单。双击“在这里输入”，将其更改为“文件(`&F`)”，然后按下回车键，效果如下图所示。这里的`&F`表明将菜单的快捷键设置为了Alt+ F，可以看到，实际的显示效果中`&`符号是隐藏的。



3．同样的方法，我们在文件菜单中添加“新建(&N)”子菜单，效果如下图所示。菜单后面的那个加号图标是用来创建下一级菜单的。



二、添加菜单图标

1．Qt中的一个菜单被看做是一个 `Action` ，我们在下面的 `Action` 编辑器中可以看到刚才添加的“新建”菜单，如下图所示。



2．双击该条目，会弹出编辑动作对话框，这里可以进行各项设置，比如我们可以设置菜单的快捷键，点击一下快捷键后面的行编辑器，然后按下键盘上的 `Ctrl + N` ，这样就可以将该菜单的快捷键设置为 `Ctrl + N` 。如下图所示。那么大家可能会问，既然该菜单的快捷键是这么设置的，那么前面设置的“新建(N)”中的N是什么呢？这个可以被称为加速键，就是只有当文件菜单处于激活（显示）状态时，按下N键才会执行新建菜单的功能。

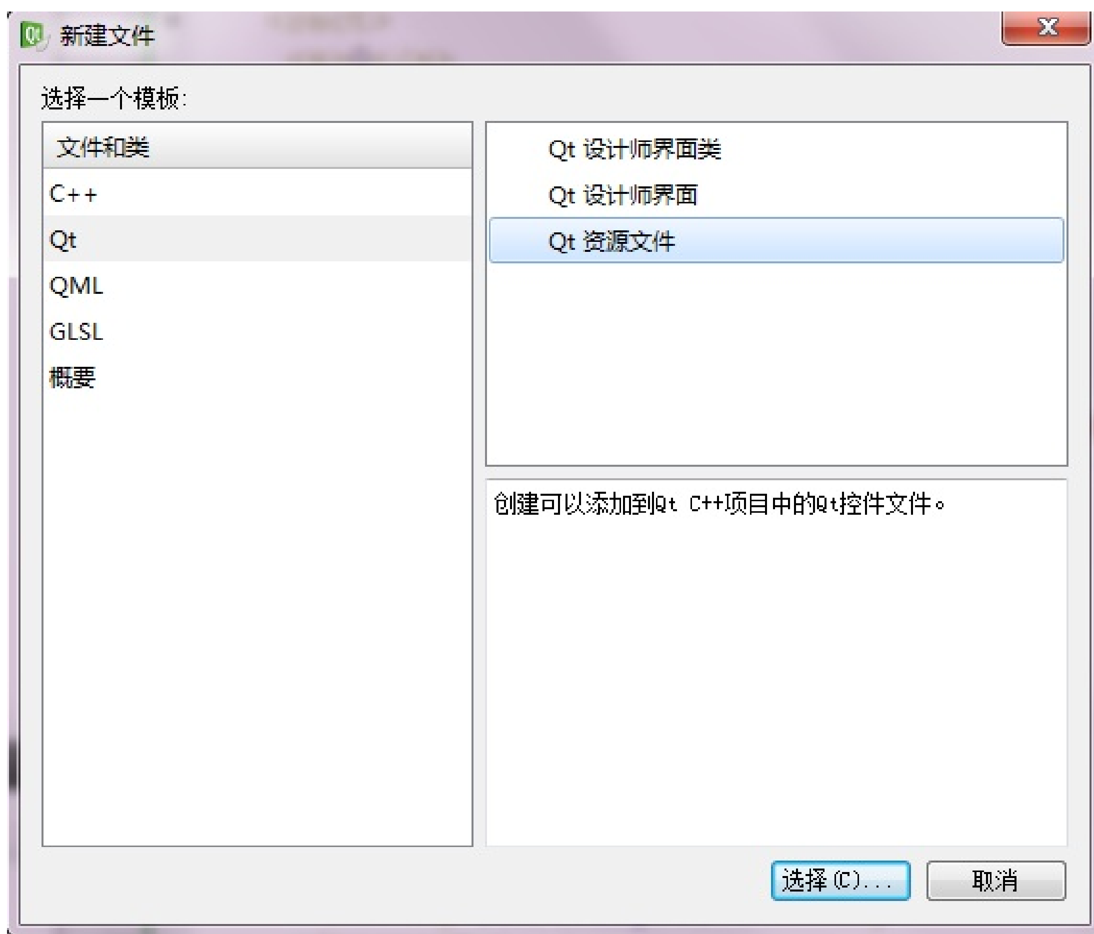


3 · 在编辑动作对话框中的图标后面的 黑色箭头下拉框可以选择使用资源还是使用文件，如果使用文件的话，那么就可以直接在弹出的文件对话框中选择本地磁盘上的一个图标文件。下面我们来讲述使用资源的方式，如果直接点击这个按钮就是默认的使用资源。现在我们先按下编辑动作对话框的确定按钮关闭它。

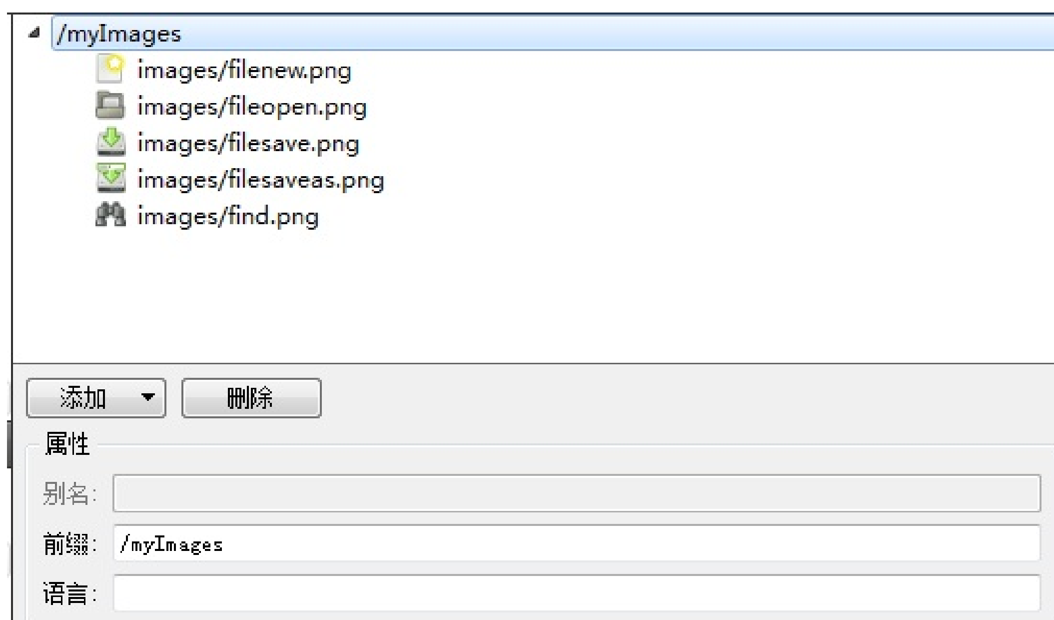
三、添加资源文件

1 · Qt中可以使用资源文件将各种类型的文件添加到最终生成的可执行文件中，这样就可以避免使用外部文件可能出现的一些问题。而且，在编译时Qt还会将资源文件进行压缩，我们可能发现生成的可执行文件比我们添加到其中的资源文件还要小。

2 · 我们向项目中添加新文件，模板选择Qt资源文件。如下图所示。然后将名称设置为 `myResources` 。



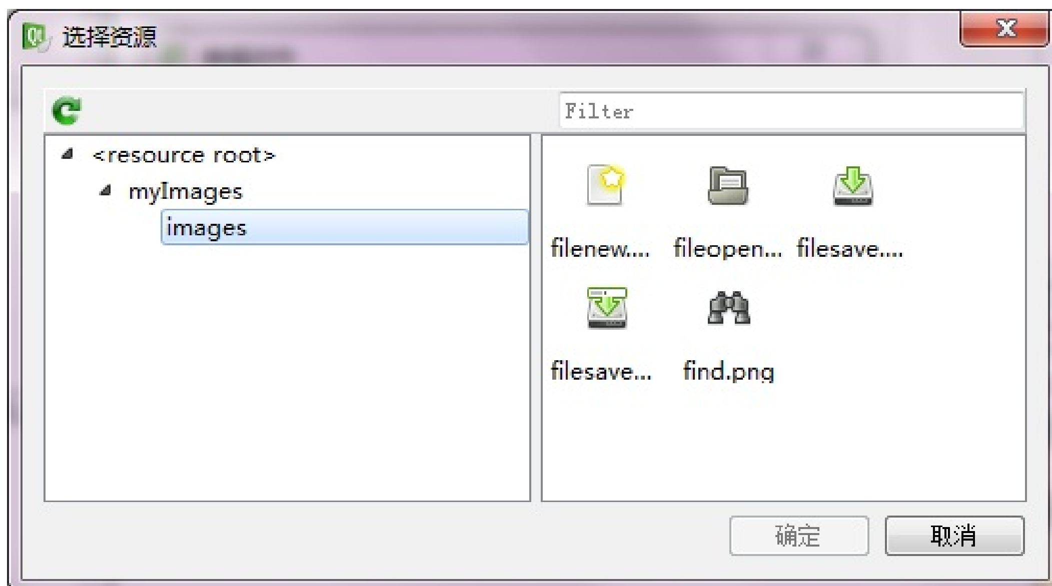
3．创建完文件后会自动打开该资源文件，这里需要先在下面添加前缀，就是点击添加按钮，然后选择前缀，默认的前缀是 `/new/prefix1` ，这个可以随意修改（不要出现中文字符），我们这里因为要添加图片，所以修改为/myImages。然后再按下添加按钮来添加文件，这里最好将所有要用到的图片放到项目目录中。比如我们这里在项目目录中新建了一个 `images` 文件夹，然后将需要的图标文件粘贴进去。添加完文件后，如下图所示。



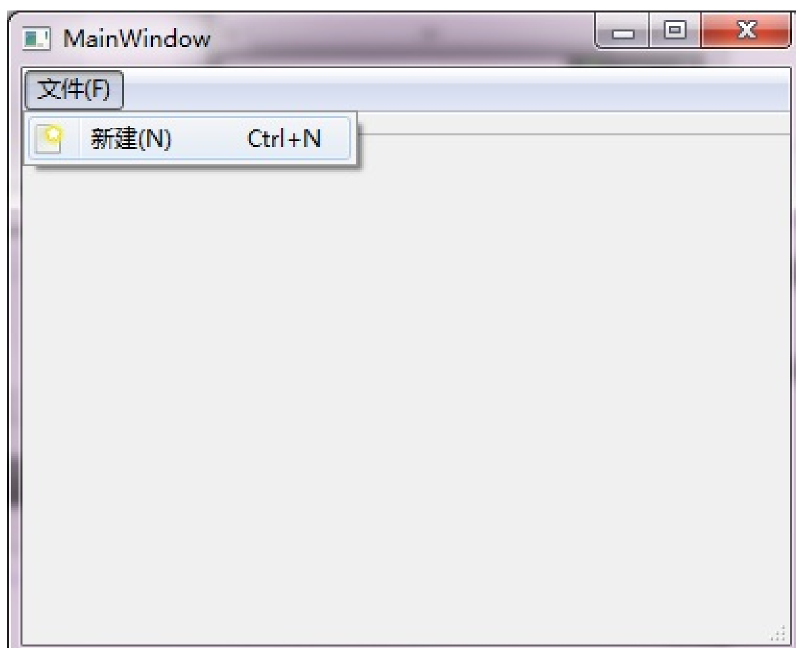
4．当添加完资源后，一定要按下 `Ctrl + S` 来保存资源文件，不然在后面可能无法显示已经添加的资源。

四、使用资源文件

1．我们重新到设计模式打开新建菜单的编辑动作对话框，然后添加图标。在打开的选择资源对话框中，第一次可能无法显示已经存在的资源，可以按下左上角的绿箭头来更新显示。效果如下图所示。



2．我们点击这里需要的新建图标 `filenew.png` ，按下确定即可。现在按下 `Ctrl + R` 键运行程序，效果如下图所示。



五、使用代码来添加菜单和图标

1．对于添加的资源文件，在项目目录中可以看到，即 `myResources.qrc` ，使用写字板程序将其打开，可以发现它其实就是一个XML文档：

```
<RCC>
    <qresourceprefix="/myImages">
       <file>images/filenew.png</file>
       <file>images/fileopen.png</file>
       <file>images/filesave.png</file>
       <file>images/filesaveas.png</file>
       <file>images/find.png</file>
    </qresource>
</RCC>
```

2. 前面是在设计模式添加的图标文件，下面我们使用代码再来添加一个菜单，并为其设置图标。在编辑模式打开 `mainwindow.cpp` 文件，并在构造函数中添加如下代码：

```
// 创建新的动作
QAction *openAction = new QAction(tr("&Open"), this);
// 添加图标
QIcon icon(":/myImages/images/fileopen.png");
openAction->setIcon(icon);
// 设置快捷键
openAction->setShortcut(QKeySequence(tr("Ctrl+O")));
// 在文件菜单中设置新的打开动作
ui->menu_F->addAction(openAction);
```

这里添加图标时，就是使用的资源文件中的图标。使用资源文件，需要在最开始使用冒号，然后添加前缀，后面是文件的路径。在代码中使用文件菜单，就是使用其 `objectName` 。大家现在可以运行程序查看效果，当然这里也可以将Open改为中文。

结语

这一篇中主要讲解了如何使用资源文件，讲述了在设计模式和代码中两种使用方法。希望大家可以亲自练习一下本篇的内容，在后面的章节中，对于添加菜单和图标等操作将不再进行讲解。

涉及到的源码下载

图标文件集合

第5篇 基础（五）Qt布局管理器

导语

在前一篇中我们学习了使用资源文件为主窗口添加菜单图标。这次，我们先将菜单进行完善，然后讲解一些布局管理方面的内容。一个软件不仅要有强大的功能，还要有一个美观的界面，布局管理器就是用来对界面部件进行布局管理的。这一节将简单介绍一下Qt的布局方面的应用，大家可以以此类推，学习使用其他布局部件。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

目录

正文

一、完善菜单

1．新建Qt Gui应用，项目名称为 `myMainWindow` ，基类选择 `QMainWindow` ，类名为 `MainWindow` 。

2．完成后，在设计模式添加菜单项，并添加资源文件，向其中添加菜单图标。最终各个菜单如下图所示。



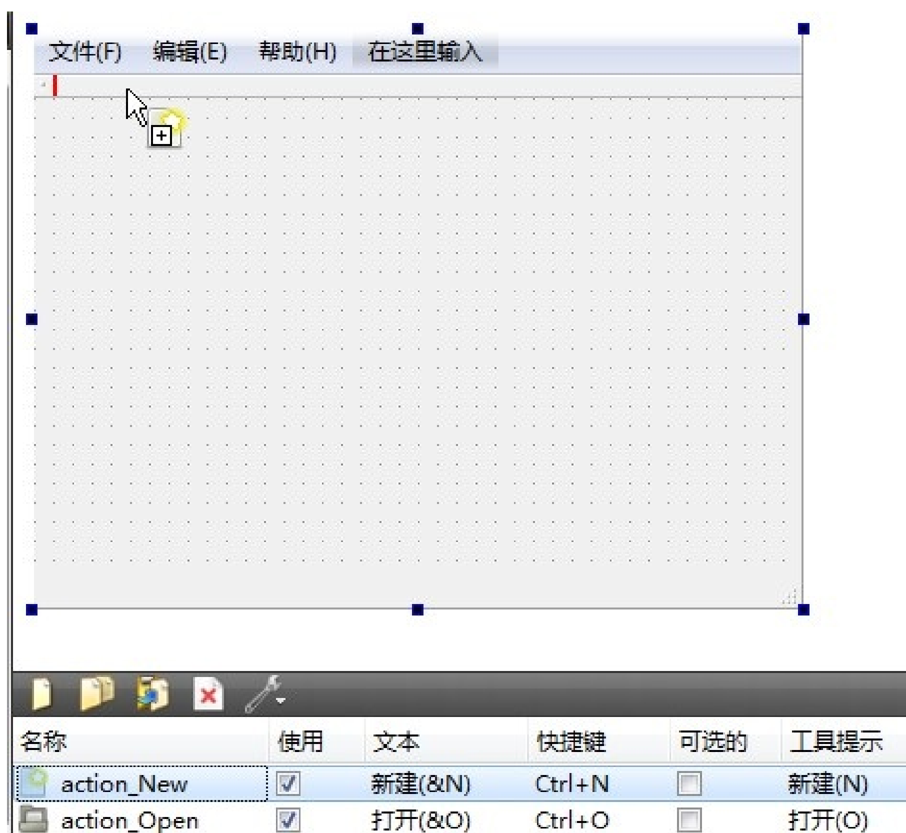


在 `Action` 编辑器中修改动作的对象名称、图标和快捷键，最终如下图所示。

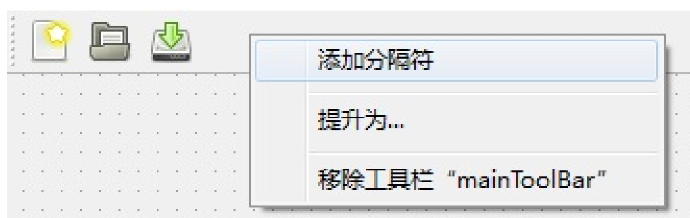
名称	使用	文本	快捷键	可选的	工具提示
action_New	☑	新建(&N)	Ctrl+N	☐	新建(N)
action_Open	☑	打开(&O)	Ctrl+O	☐	打开(O)
action_Close	☑	关闭(&C)	Ctrl+W	☐	关闭(C)
action_Save	☑	保存(&S)	Ctrl+S	☐	保存(S)
action_SaveAs	☑	另存为(&A)		☐	另存为(A)
action_Exit	☑	退出(&X)	Ctrl+Q	☐	退出(X)
action_Undo	☑	撤销(&Z)	Ctrl+Z	☐	撤销(Z)
action_Cut	☑	剪切(&X)	Ctrl+X	☐	剪切(X)
action_Copy	☑	复制(&C)	Ctrl+C	☐	复制(C)
action_Paste	☑	粘贴(&V)	Ctrl+V	☐	粘贴(V)
action_Find	☑	查找(&F)	Ctrl+F	☐	查找(F)
action_Help	☑	版本说明	Ctrl+H	☐	版本说明

二、向工具栏添加菜单图标

可以将动作编辑器中的动作拖动到工具栏中作为快捷图标使用，如下图所示。



可以在工具栏上点击鼠标右键来添加分隔符，如下图所示。

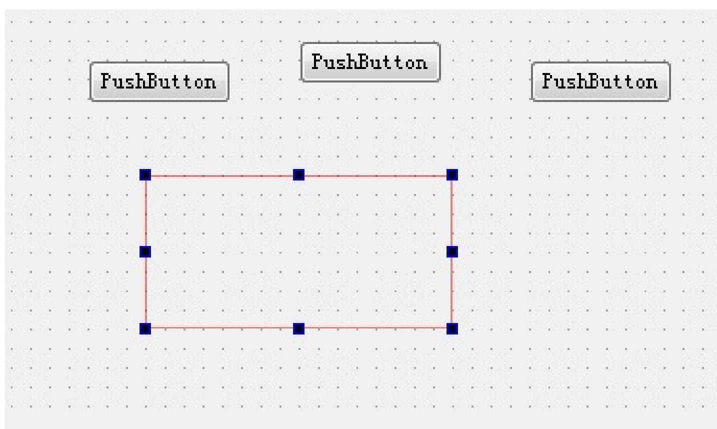


最终工具栏如下图所示。

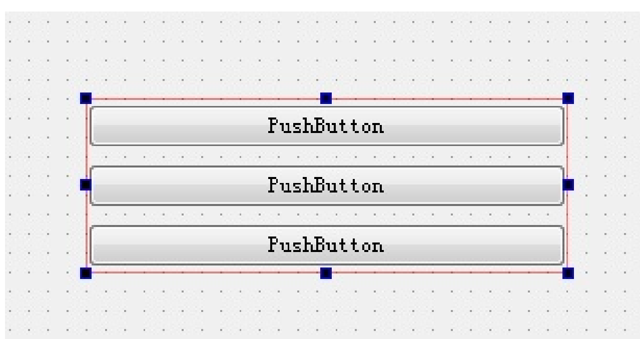


三、布局管理器

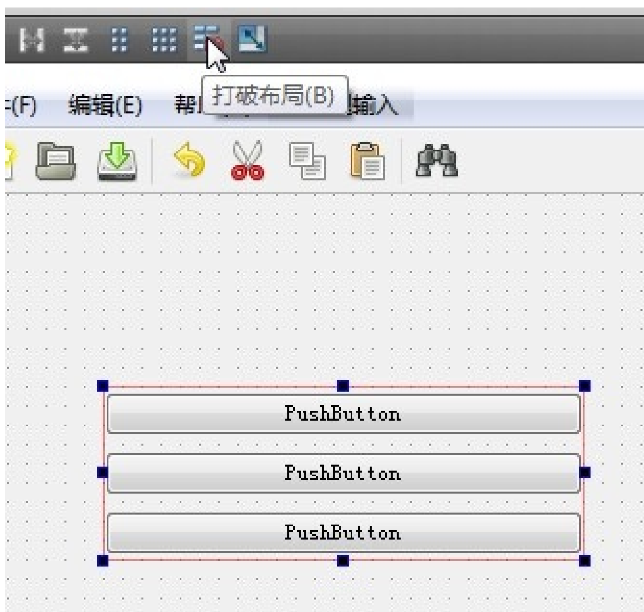
1．从左边控件栏中拖入三个按钮 `Push Button` 和一个 `Vertical Layout` （垂直布局管理器）到界面上，如下图所示。



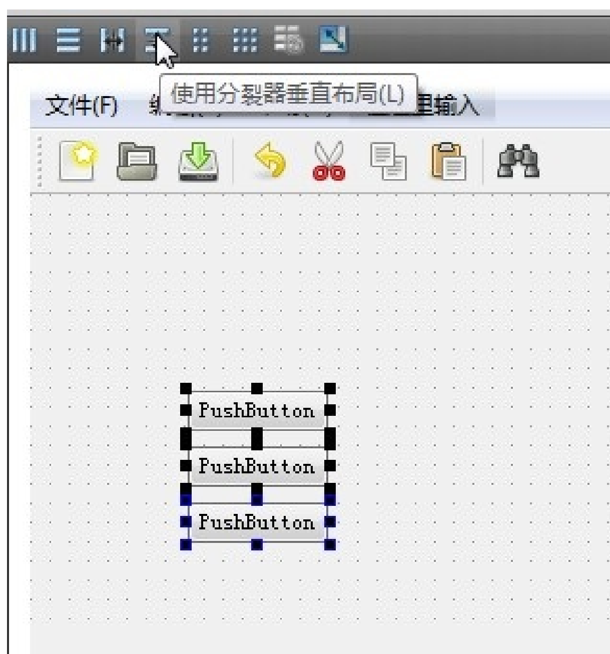
然后将三个按钮拖入到布局管理器中，这时三个按钮就会自动垂直排列，并且进行水平拉伸，无论如何改变布局管理器的大小，按钮总是水平方向变化。如下图所示。



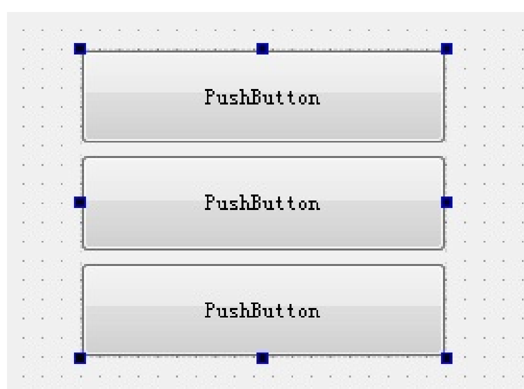
2．我们可以选中布局管理器，然后按下上方工具栏中的“打破布局”按钮来删除布局管理器。（当然也可以先将三个按钮移出，然后按下 `Delete` 键来删除布局管理器，如果不移出按钮，那么会将它们同时删除。）如下图所示。



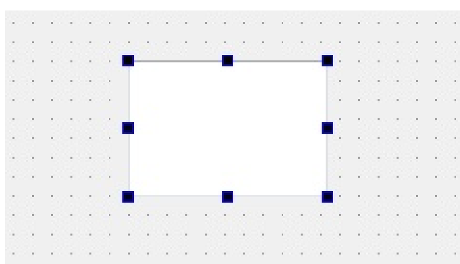
3．下面我们使用分裂器（ `QSplitter` ）来进行布局，先同时选中三个按钮，然后按下上方工具栏中的“使用分裂器垂直布局”按钮，如下图所示。



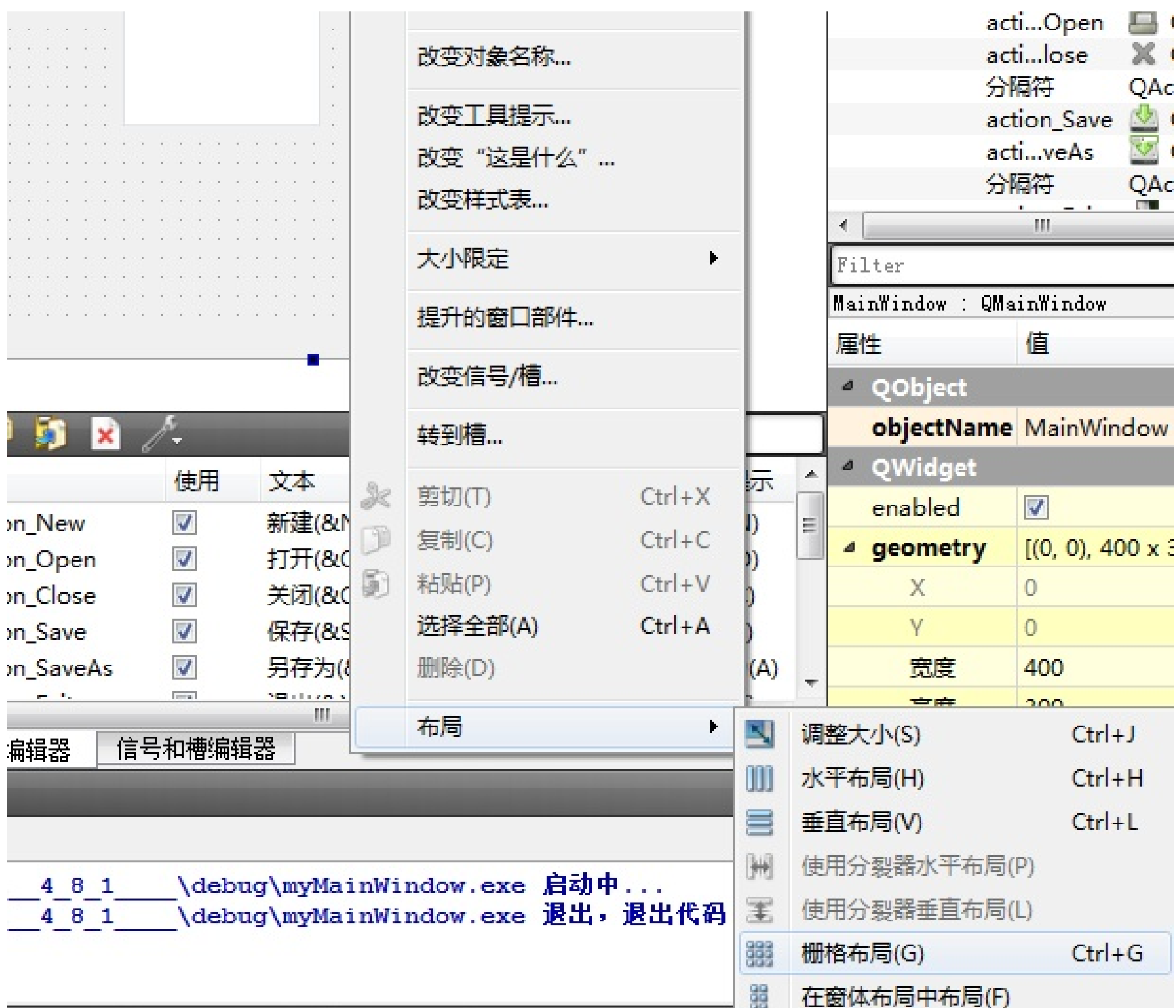
然后我们进行放大，可以发现，使用分裂器按钮纵向是可以变大的，这就是分裂器和布局管理器的重要区别。如下图所示。



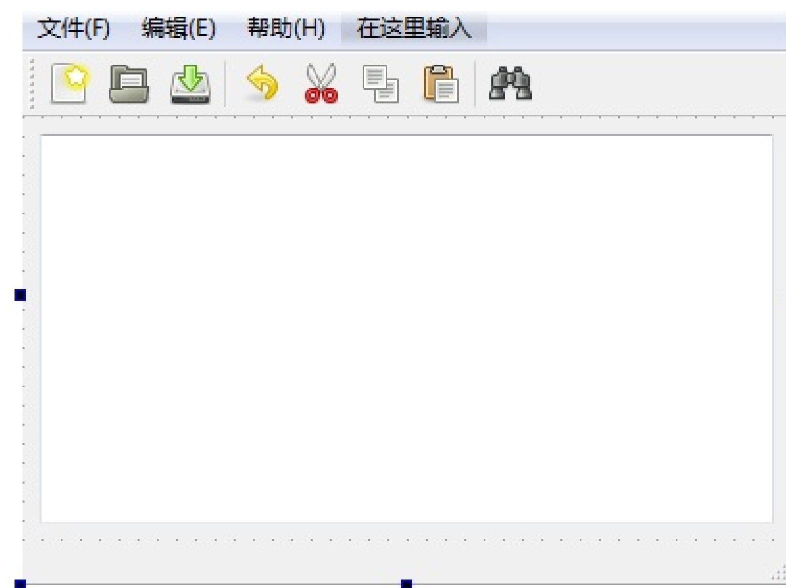
4．布局管理器除了可以对部件进行布局以外，还有个重要用途，就是使部件随着窗口的大小变化而变化。我们删除界面上的部件，然后拖入一个文本编辑器 `Text Edit` 部件。如下图所示。



然后我们在界面上点击鼠标右键，选择布局→栅格布局（或者使用快捷键 `Ctrl+G` ）。



这时整个文本编辑器部件就会填充中央区域。如下图所示。现在运行程序，可以发现，无论怎样拉伸窗口，文本编辑器总是填充整个中央区域。



这一篇中我们主要介绍了布局管理器的应用，而且都是在设计模式对布局管理器的使用。这里使用三种方式进行了演示：从控件栏中拖入布局管理器；在工具栏中使用图标；还有使用右键菜单（当然还有快捷键的方式）。

结语

在本篇中主要对垂直布局管理器进行了演示，大家还可以使用其他的布局管理器进行测试。对于如何在代码中使用布局管理器，以及使用布局管理器实现可隐藏窗口，可以参考《Qt Creator快速入门》一书的相关章节。

涉及到的源码下载

第6篇 基础（六）实现Qt文本编辑功能

导语

前面已经在主窗口中添加了菜单和工具栏，这一篇中我们将实现基本的文本编辑功能。在开始正式写程序之前，我们先要考虑一下整个流程。因为这里要写一个记事本一样的程序，所以最好先打开Windows中的记事本，进行一些简单的操作，然后考虑怎样去实现这些功能。再者，再强大的软件，它的功能也是一个一个加上去的，不要设想一下子写出所有的功能。我们这里先实现新建文件，保存文件，和文件另存为三个功能，是因为它们联系很紧，而且这三个功能总的代码量也不是很大。

环境是：Windows 7 + Qt 4.8.1+ Qt Creator 2.4.1

目录

正文

一、实现新建文件、文件保存和另存为功能

1．首先来分析下整个流程，当新建文件时，要考虑是否保存正在编辑的文件，如果需要保存，还要根据该文件以前是否保存过来进行保存或者另存为操作。下面我们根据这里的分析来添加需要的函数和对象。

2．打开上一篇完成的项目，然后先在 `main.cpp` 文件中添加代码来保证代码中可以使用中文字符。 首先添加 `#include <QTextCodec>` 头文件包含，然后在主函数中添加如下代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

3．在 `mainwindow.h` 文件中添加 `public` 函数声明：

```
void newFile();    // 新建操作
bool maybeSave(); // 判断是否需要保存
bool save();       // 保存操作
bool saveAs();     // 另存为操作
bool saveFile(const QString &fileName); // 保存文件
```

这里的几个函数就是用来完成功能逻辑的，下面我们会添加它们的定义来实现相应的功能。因为这几个功能联系紧密，所以这几个函数会相互调用。

4．然后添加 `private` 变量定义：

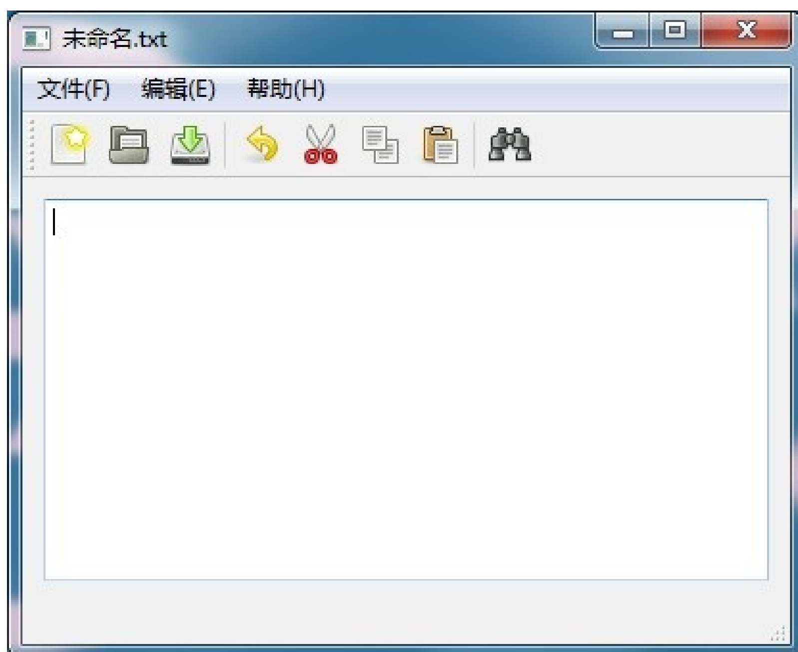
```
// 为真表示文件没有保存过，为假表示文件已经被保存过了
bool isUntitled;
// 保存当前文件的路径
QString curFile;
```

这里的 `isUntitled` 是一个标志，用来判断文档是否被保存过。而 `curFile` 用来保存当前打开的文件的路径。

5．下面到 `mainwindow.cpp` 文件，先添加头文件：

```
#include <QMessageBox>
#include <QPushButton>
#include <QFileDialog>
#include <QTextStream>
然后在构造函数中添加如下代码来进行一些初始化操作：
// 初始化文件为未保存状态
isUntitled = true;
// 初始化文件名为"未命名.txt"
curFile = tr("未命名.txt");
// 初始化窗口标题为文件名
setWindowTitle(curFile);
```

这里设置了在启动程序时窗口标题显示文件的名字，效果如下图所示。



6．下面添加那几个函数的定义。

首先是新建文件操作的函数：

```
void MainWindow::newFile()
{
   if (maybeSave()) {
       isUntitled = true;
       curFile = tr("未命名.txt");
       setWindowTitle(curFile);
       ui->textEdit->clear();
       ui->textEdit->setVisible(true);
   }
}
```

这里先使用 `maybeSave()` 来判断文档是否需要保存，如果已经保存完了，则新建文档，并进行初始化。下面是 `maybeSave()` 函数的定义：

```
bool MainWindow::maybeSave()
{
    // 如果文档被更改了
    if (ui->textEdit->document()->isModified()) {
    // 自定义一个警告对话框
       QMessageBox box;
       box.setWindowTitle(tr("警告"));
       box.setIcon(QMessageBox::Warning);
       box.setText(curFile + tr(" 尚未保存，是否保存？"));
       QPushButton *yesBtn = box.addButton(tr("是(&Y)"),
                        QMessageBox::YesRole);
       box.addButton(tr("否(&N)"), QMessageBox::NoRole);
       QPushButton *cancelBut = box.addButton(tr("取消"),
                        QMessageBox::RejectRole);
       box.exec();
       if (box.clickedButton() == yesBtn)
            return save();
       else if (box.clickedButton() == cancelBut)
            return false;
   }
   // 如果文档没有被更改，则直接返回true
   return true;
}
```

这里先使用了 `isModified()` 来判断文档是否被更改了，如果被更改了，则弹出对话框让用户选择是否进行保存，或者取消操作。如果取消操作，那么就返回 `false` ，什么都不执行。下面是 `save()` 函数的定义：

```
bool MainWindow::save()
{
   if (isUntitled) {
       return saveAs();
   } else {
       return saveFile(curFile);
   }
}
```

这里如果文档以前没有保存过，那么执行另存为操作 `saveAs()` ，如果已经保存过，那么调用 `saveFile()` 执行文件保存操作。下面是 `saveAs()` 函数的定义：

```
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                            tr("另存为"),curFile);
    if (fileName.isEmpty()) return false;
    return saveFile(fileName);
}
```

这里使用 `QFileDialog` 来实现了一个另存为对话框，并且获取了文件的路径，然后使用文件路径来保存文件。下面是 `saveFile()` 函数的定义：

```
bool MainWindow::saveFile(const QString &fileName)
{
    QFile file(fileName);

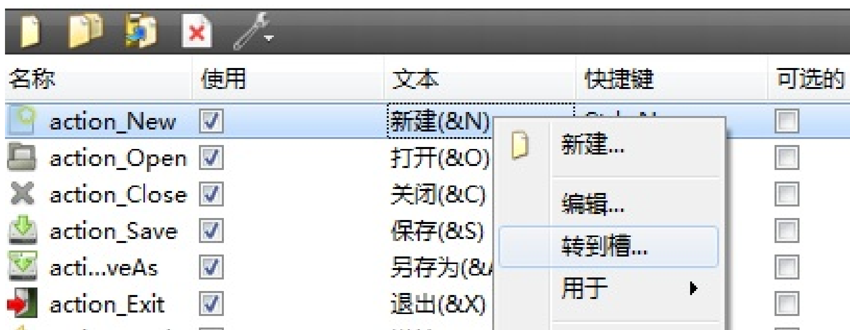
    if (!file.open(QFile::WriteOnly | QFile::Text)) {

        // %1和%2分别对应后面arg两个参数，/n起换行的作用
        QMessageBox::warning(this, tr("多文档编辑器"),
                    tr("无法写入文件 %1：/n %2")
                   .arg(fileName).arg(file.errorString()));
        return false;
    }
    QTextStream out(&file);
    // 鼠标指针变为等待状态
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << ui->textEdit->toPlainText();
    // 鼠标指针恢复原来的状态
    QApplication::restoreOverrideCursor();
    isUntitled = false;
    // 获得文件的标准路径
    curFile = QFileInfo(fileName).canonicalFilePath();
    setWindowTitle(curFile);
    return true;
}
```

该函数执行真正的文件保存操作。先是使用一个 `QFile` 类对象来指向要保存的文件，然后将其使用写入方式打开。打开后再使用 `QTextStream` 文本流将编辑器中的内容写入到文件中。

这里使用了很多新的类，以后我们对自己不明白的类都可以去帮助里进行查找，这也许是我们以后要做的最多的一件事了。对于其中的英文解释，我们最好想办法弄明白它的大意，其实网上也有一些中文的翻译，但最好还是从一开始就尝试着看英文原版的帮助，这样以后才不会对中文翻译产生依赖。

7．设置菜单功能。双击 `mainwindow.ui` 文件，在图形界面窗口下面的 `Action` 编辑器里，我们右击“新建”菜单一条，选择“转到槽”，然后选择 `triggered()` ，进入其触发事件槽。如下图所示。



同理，进入其他两个菜单的槽，将相应的操作的函数写入槽中。最终代码如下：

```
void MainWindow::on_action_New_triggered()
{
   newFile();
}
void MainWindow::on_action_Save_triggered()
{
   save();
}
void MainWindow::on_action_SaveAs_triggered()
{
   saveAs();
}
```

现在运行程序，已经能够实现新建文件，保存文件，文件另存为的功能了。

二、实现打开、关闭、退出、撤销、复制、剪切、粘贴等功能

先到 `mainwindow.h` 文件中添加 `public` 函数声明：

```
bool loadFile(const QString &fileName); // 加载文件
```

然后到 `mainwindow.cpp` 文件中添加该函数的定义：

```
bool MainWindow::loadFile(const QString &fileName)
{
   QFile file(fileName); // 新建QFile对象
   if (!file.open(QFile::ReadOnly | QFile::Text)) {
       QMessageBox::warning(this, tr("多文档编辑器"),
                             tr("无法读取文件 %1:\n%2.")
                             .arg(fileName).arg(file.errorString()));
       return false; // 只读方式打开文件，出错则提示，并返回false
   }
   QTextStream in(&file); // 新建文本流对象
   QApplication::setOverrideCursor(Qt::WaitCursor);
   // 读取文件的全部文本内容，并添加到编辑器中
   ui->textEdit->setPlainText(in.readAll());      QApplication::restoreOverrideCursor(
);

   // 设置当前文件
   curFile = QFileInfo(fileName).canonicalFilePath();
   setWindowTitle(curFile);
   return true;
}
```

这里的操作和 `saveFile()` 函数是相似的。下面到设计模式，分别进入其他几个动作的触发信号的槽，更改如下：

```
// 打开动作
void MainWindow::on_action_Open_triggered()
{
    if (maybeSave()) {

        QString fileName = QFileDialog::getOpenFileName(this);

        // 如果文件名不为空，则加载文件
        if (!fileName.isEmpty()) {
            loadFile(fileName);
            ui->textEdit->setVisible(true);
        }
    }
}
// 关闭动作
void MainWindow::on_action_Close_triggered()
{
    if (maybeSave()) {
        ui->textEdit->setVisible(false);
    }
}
// 退出动作
void MainWindow::on_action_Exit_triggered()
{
    // 先执行关闭操作，再退出程序
    // qApp是指向应用程序的全局指针
    on_action_Close_triggered();
    qApp->quit();
}
// 撤销动作
void MainWindow::on_action_Undo_triggered()
{
    ui->textEdit->undo();
}
// 剪切动作
void MainWindow::on_action_Cut_triggered()
{
    ui->textEdit->cut();
}
// 复制动作
void MainWindow::on_action_Copy_triggered()
{
    ui->textEdit->copy();
}
// 粘贴动作
void MainWindow::on_action_Paste_triggered()
{
    ui->textEdit->paste();
}
```

这里可以看到，复制、粘贴等常用功能是QTextEdit已经实现的，我们只需要调用相应的函数。虽然实现了退出功能，但是，有时候会使用窗口标题栏的关闭按钮来关闭程序，这里我们需要使用关闭事件处理函数来实现相应的功能。

下面到mainwindow.h文件中，先添加头文件包含 `#include <QCloseEvent>` ，然后添加函数声明：

```
protected:
    void closeEvent(QCloseEvent *event); // 关闭事件

然后到mainwindow.cpp文件中添加该函数的定义：
void MainWindow::closeEvent(QCloseEvent *event)
{
   // 如果maybeSave()函数返回true，则关闭程序
   if (maybeSave()) {
       event->accept();
   } else {   // 否则忽略该事件
       event->ignore();
   }
}
```

关于事件的概念，会在后面的教程中讲解。

结语

这一篇中实现了最基本的编辑功能，现在还剩下查找和帮助菜单没有实现，这个会在下一篇进行介绍。如果大家想学习一个更完整的文本编辑器的实现，可以参考《Qt及QtQuick开发实战精解》一书的第一章。

涉及到的源码下载

第7篇 基础（七）实现Qt文本查找功能

导语

这一篇我们来加上查找菜单的功能。因为要涉及Qt Creator的很多实用功能，所以单独用一篇文章来介绍。以前都用设计器设计界面，而这次我们用代码实现一个简单的查找对话框。除了讲解怎么实现查找功能，这里还详细地说明了怎么进行类中方法的查找和使用。其中也讲解了Qt Creator程序中怎样在函数的声明位置和定义位置间进行快速切换。

环境是：Windows 7 + Qt 4.8.1+ Qt Creator 2.4.1

目录

一、添加查找对话框 二、实现查找功能

正文

一、添加查找对话框

1．我们继续在前一篇程序的基础之上进行更改。首先到 `mainwindow.h` 文件中添加类的前置声明（对于什么是前置声明，以及这样使用的好处，可以在网上百度）：

```
class QLineEdit;
class QDialog;
```

前置声明所在的位置跟头文件包含的位置相同。

然后在 `private` 中添加对象定义：

```
QLineEdit *findLineEdit;
QDialog *findDlg;
```

下面再添加一个私有槽声明：

```
private slots:
    void showFindText();
```

槽可以看做是一个函数，只不过可以和信号进行关联。

2．下面到 `mainwindow.cpp` 文件中，因为前面在头文件中使用了类的前置声明，所以这里需要先添加头文件包含：

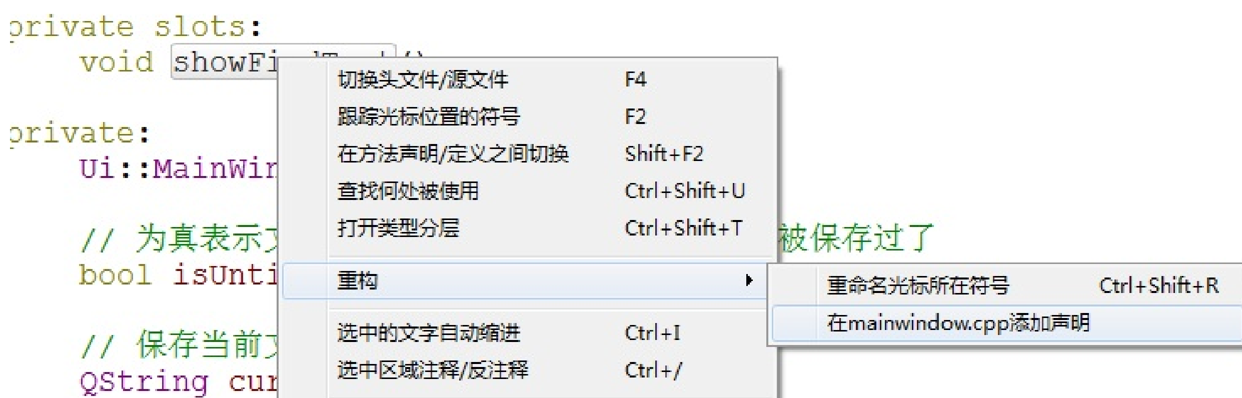
```
#include <QLineEdit>
#include <QDialog>
#include <QPushButton>
```

然后在构造函数中进行初始化操作，即添加如下代码：

```
findDlg = new QDialog(this);
findDlg->setWindowTitle(tr("查找"));
findLineEdit = new QLineEdit(findDlg);
QPushButton *btn= new QPushButton(tr("查找下一个"), findDlg);
QVBoxLayout *layout= new QVBoxLayout(findDlg);
layout->addWidget(findLineEdit);
layout->addWidget(btn);
connect(btn, SIGNAL(clicked()), this, SLOT(showFindText()));
```

这里创建了一个对话框，然后将一个行编辑器和一个按钮放到了上面，并使用布局管理器进行布局。最后将按钮的单击信号关联到了自定义的显示查找到的文本槽上。下面来添加该槽的定义。

3．这里先说一个可以快速从头文件声明处创建函数定义的方法。到 `mainwindow.h` 文件中，将鼠标定位到 `showFindText()` 函数上，然后点击右键，在弹出的菜单中选择“重构”→“在 `mainwindow.cpp` 添加声明”，或者直接使用 `Alt+Enter` 快捷键，这样就会直接在 `mainwindow.cpp` 文件中添加函数定义，并跳转到该函数处。如下图所示。



二、实现查找功能

下面我们来分步骤完成 `showFindText()` 函数。在讲解过程中会涉及一些很实用的功能的介绍。

1．先在函数中添加一行代码来获取行编辑器中要查找的字符串。

```
void MainWindow::showFindText()
{
   QString str = findLineEdit->text();
}
```

2．在下一行，我们先输入 `ui` ，然后按下键盘上的 `>.` 键，这时就会自动输入 `.` 或者 `->` ，并且列出 `ui` 上所有可用部件的对象名。如下图所示。

```
void MainWindow::showFindText()
{
    QString str = findLineEdit->text();
    ui->
}       action_Close
        action_Copy
        action_Cut
        action_Exit
        action_Find
```

3 · 我们要输入 `textEdit` ，先输入 `t` ，这时会自动弹出 `textEdit` ，只需要按下回车键即可。如下图所示。

```
void MainWindow::showFindText()
{
    QString str = findLineEdit->text();
    ui->t
}       textEdit
```

4 · 下面我们将光标放到 `textEdit` 上，这时就会出现 `QTextEdit` 类的简单介绍，如下图所示。



5 · 按照提示，我们按下键盘上的 `F1` 键，就会在编辑器的右侧打开 `QTextEdit` 类的帮助文档。如下图所示。这时还可以按下上面的“切换至帮助模式”来进入到帮助模式中打开该文档。



6 · 我们在该类的 `Public Functions` 公共函数列表中发现有一个 `find()` 函数。如下图所示。

QList<ExtraSelection>	**extraSelections** () const
bool	**find** (const QString & *exp*, QTextDocument::FindFlags *options* = 0)
QString	**fontFamily** () const

7、从字面意思上可以知道该函数应该是用于查找功能的，我们点击该函数进入到它的详细介绍处。如下图所示。

> bool QTextEdit::find (const QString & *exp*, QTextDocument::FindFlags *options* = 0)
>
> Finds the next occurrence of the string, *exp*, using the given *options*. Returns true if *exp* was found and changes the cursor to select the match; otherwise returns false.

8、根据介绍可以知道该函数用于查询指定的 `exp` 字符串，如果找到了就将光标跳转到查找到的位置，如果没有找到就返回 `false` 。这个函数还有一个 `QTextDocument::FindFlags` 参数，为了了解该参数的意思，我们点击该参数进入其详细介绍处。如下图所示。

enum QTextDocument::FindFlag
flags QTextDocument::FindFlags

This enum describes the options available to QTextDocument's find function. The options can be OR-ed together from the following list:

Constant	Value	Description
QTextDocument::FindBackward	0x00001	Search backwards instead of forwards.
QTextDocument::FindCaseSensitively	0x00002	By default find works case insensitive. Specifying this option changes the behaviour to a case sensitive find operation.
QTextDocument::FindWholeWords	0x00004	Makes find match only complete words.

可以看到该参数是一个枚举变量，用来指定查找的方式，分别是向后查找、区分大小写、全词匹配等。如果不指定该参数，默认的是向前查找、不区分大小写、包含该字符串的词也可以查找到。这几个变量还可以使用 `|` 符号来一起使用。

9、根据帮助，我们补充完该行代码：

```
ui->textEdit->find(str, QTextDocument::FindBackward);
```

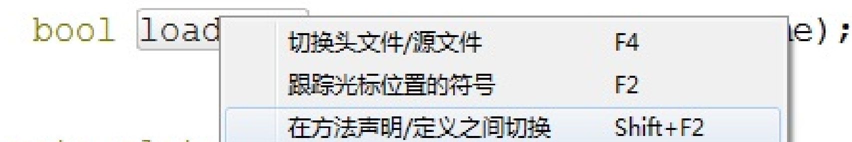
10、这时已经能实现查找的功能了。但是我们刚才看到 `find` 的返回值类型是 `bool` 型，而且，我们也应该为查找不到字符串作出提示。将这行代码更改为：

```
if (!ui->textEdit->find(str, QTextDocument::FindBackward))
{
   QMessageBox::warning(this, tr("查找"),
            tr("找不到%1").arg(str));
}
```

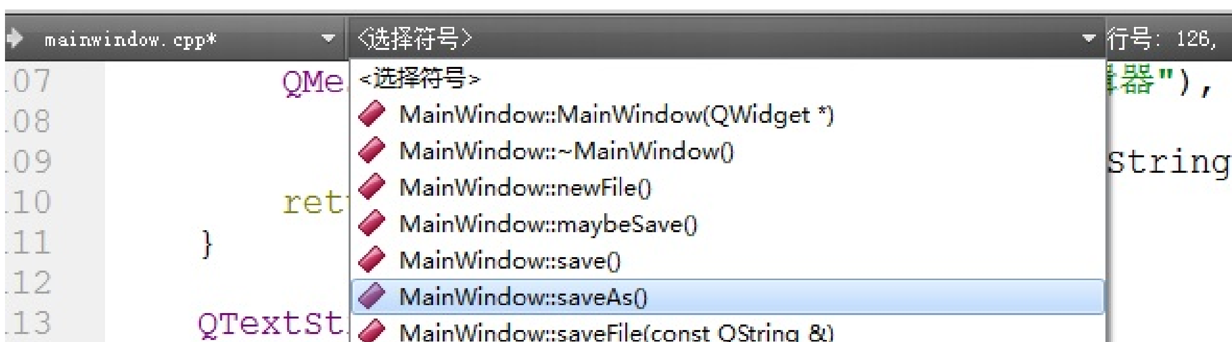
到这里查找函数就基本讲完了。

11、我们会发现随着程序功能的增强，其中的函数也会越来越多，我们都会为查找某个函数的定义位置感到头疼。而在QtCreator中有几种快速定位函数的方法。

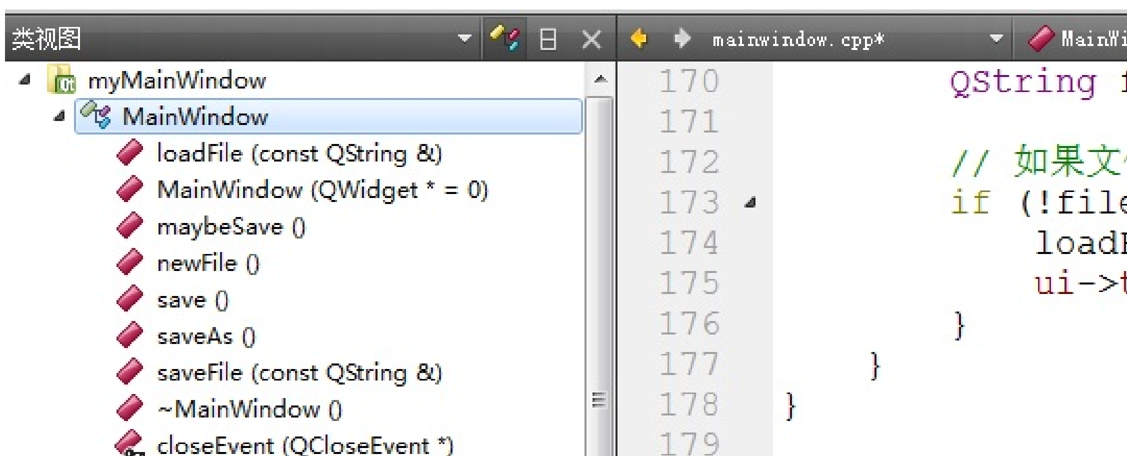
第一种，在函数声明的地方直接跳转到函数定义的地方。 例如我们在 `mainwindow.h` 文件的 `loadFile()` 函数上点击鼠标右键，在弹出的菜单上选择“在方法声明/定义之间切换”，这时就会自动跳转到 `mainwindow.cpp` 文件中该函数的定义处。如下图所示。当然还可以反向使用。



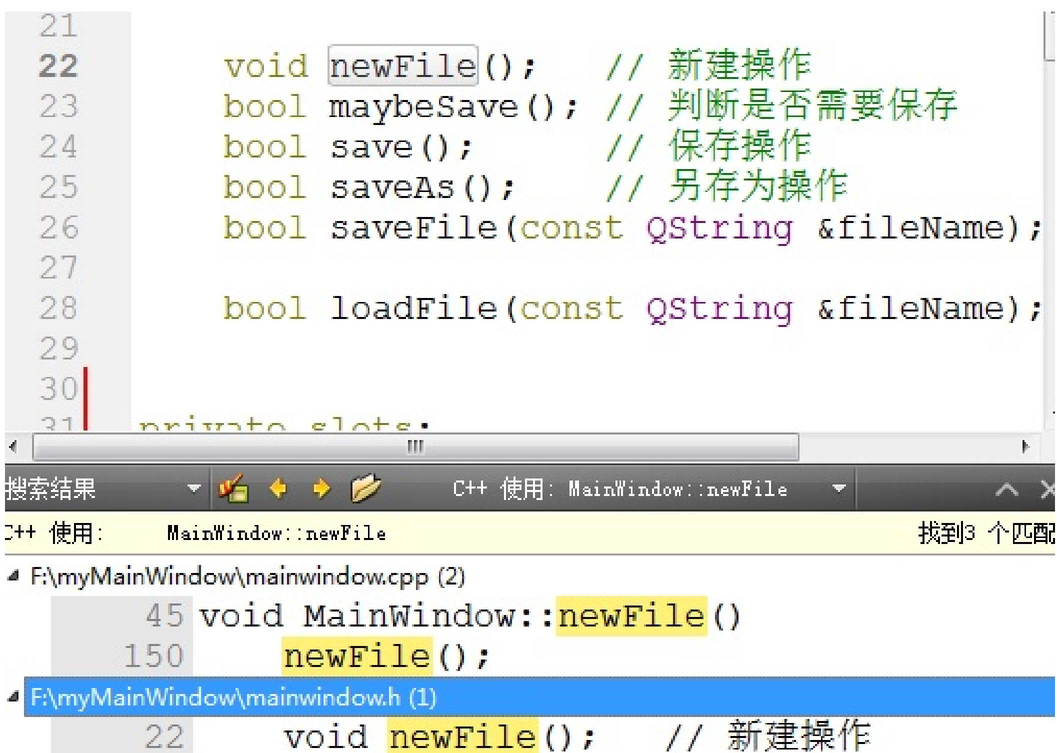
第二种，快速查看一个文件里的所有函数。 可以在编辑器正上方的下拉框里查看正在编辑的文件中所有的函数的列表，点击一个函数就会跳转到指定位置。如下图所示。



第三种，使用类视图或者大纲视图。 在项目列表上面的下拉框中可以更改查看的内容，如果选择为类视图或者大纲，则会显示文件中所有的函数的列表。如下图所示。



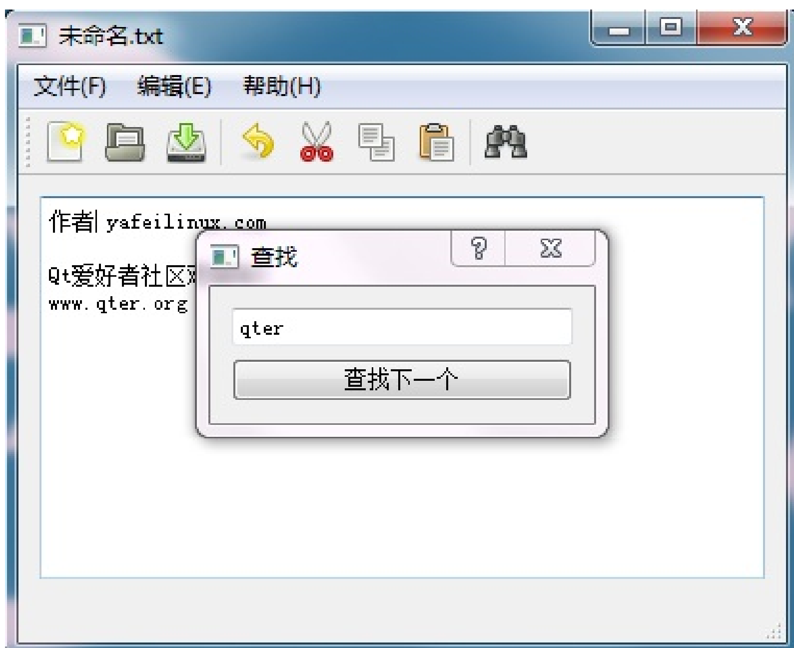
第四种，使用查找功能查看函数的所有调用处。 在一个函数名上点击鼠标右键，然后选择“查找何处被使用”菜单，这时就会在下面的搜索结果栏中显示该函数所有的使用位置。我们可以通过点击一个位置来跳转到该位置。如下图所示。



12 · 最后，我们来实现界面上的查找功能。从设计模式进入查找动作的触发信号的槽，更改如下：

```
void MainWindow::on_action_Find_triggered()
{
    findDlg->show();
}
```

这时运行程序，效果如下图所示。



结语

讲到这里，我们已经很详细地说明了怎样去使用一个类里面未接触过的函数；也说明了Qt Creator中的一些便捷操作。可以看到，Qt Creator开发环境，有很多很人性化的设计，我们应该熟练应用它们。在以后的文章中，我们不会再很详细地去用帮助来说明一个函数是怎么来的，该怎么用，这些应该自己试着去查找。

涉及到的代码

第8篇 基础（八）设置Qt状态栏

导语

在程序主窗口QMainWindow中，主要包含菜单栏，工具栏，中心部件和状态栏。前面几个已经讲过了，这一篇讲解状态栏的使用。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

目录

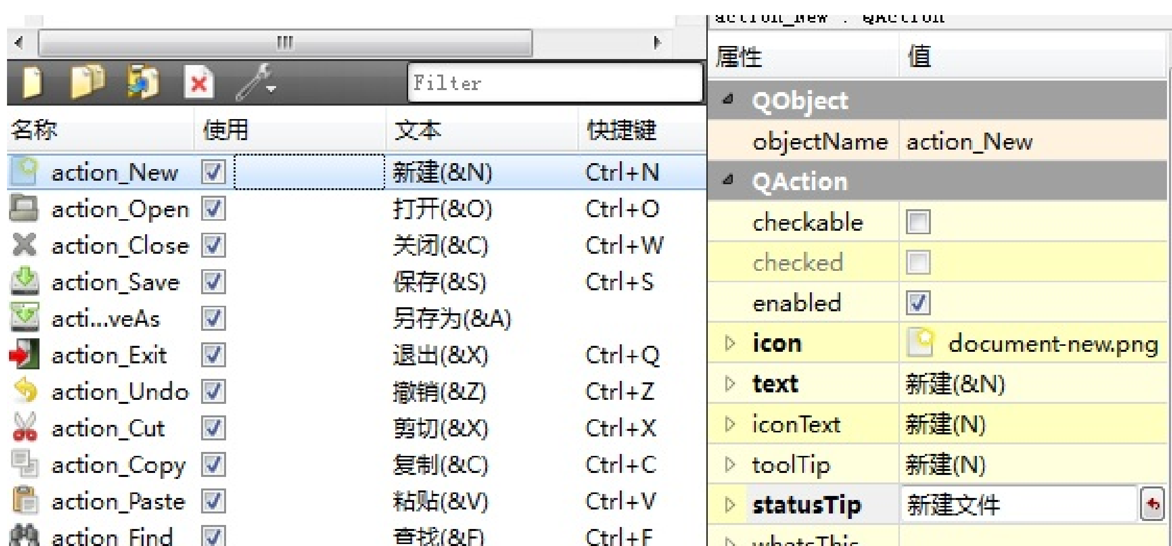
- 一、添加动作状态提示
- 二、显示其他临时信息
- 三、显示永久信息

正文

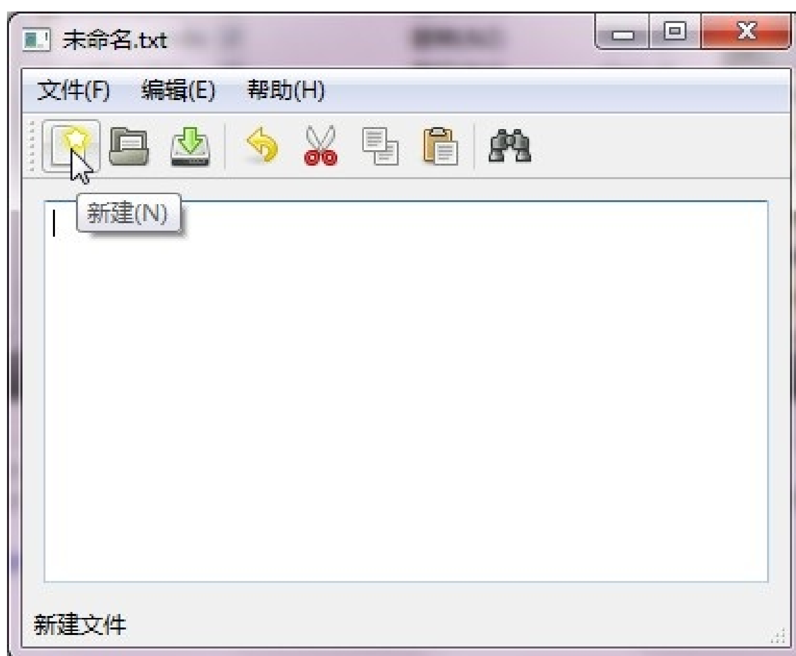
一、添加动作状态提示

1．首先还是打开上一篇完成的程序。对于菜单动作添加状态提示，可以很容易的在设计器中来完成。

2．下面进入设计模式，在Action编辑器中选中新建动作，然后在右面的属性编辑器中将其 `statusTip` 更改为“新建文件”。如下图所示。



3．这时运行程序，当光标移动到新建动作上时，在下面的状态栏将会出现设置的提示。如下图所示。



我们可以按照这种方式来设置其他动作的状态栏提示信息。

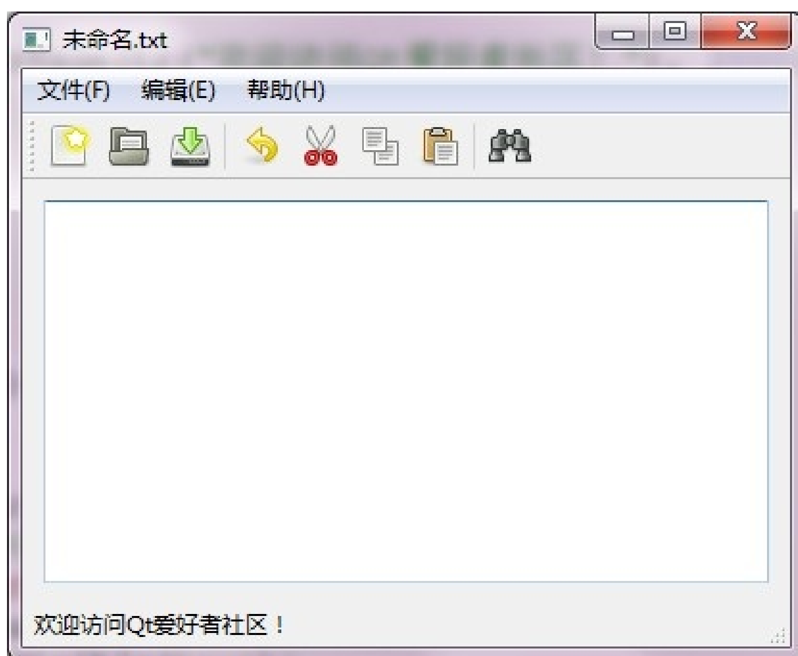
二、显示其他临时信息

状态信息可以被分为三类：临时信息，如一般的提示信息，上面讲到的动作提示就是临时信息；正常信息，如显示页数和行号；永久信息，如显示版本号或者日期。可以使用 `showMessage()` 函数来显示一个临时消息，它会出现在状态栏的最左边。一般用 `addWidget()` 函数添加一个 `QLabel` 到状态栏上用于显示正常信息，它会生成到状态栏的最左边，可能会被临时消息所掩盖。

1．我们到 `mainwindow.cpp` 文件的构造函数最后面添加如下一行代码：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区！"));
```

这样就可以在运行程序时显示指定的状态提示了。效果如下图所示。



这个提示还可以设置显示的时间。如：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区！"), 2000);
```

这样提示显示2000毫秒即2秒后会自动消失。

2．下面我们在状态栏添加一个标签部件用来显示一般的提示信息。因为无法在设计模式向状态栏添加部件，所以只能使用代码来实现。先在 `mainwindow.h` 文件中添加类的前置声明：

```
class QLabel;
```

然后添加一个私有对象定义：

```
QLabel *statusLabel;
```

下面到 `mainwindow.cpp` 文件中，先添加头文件声明：

```
#include <QLabel>
```

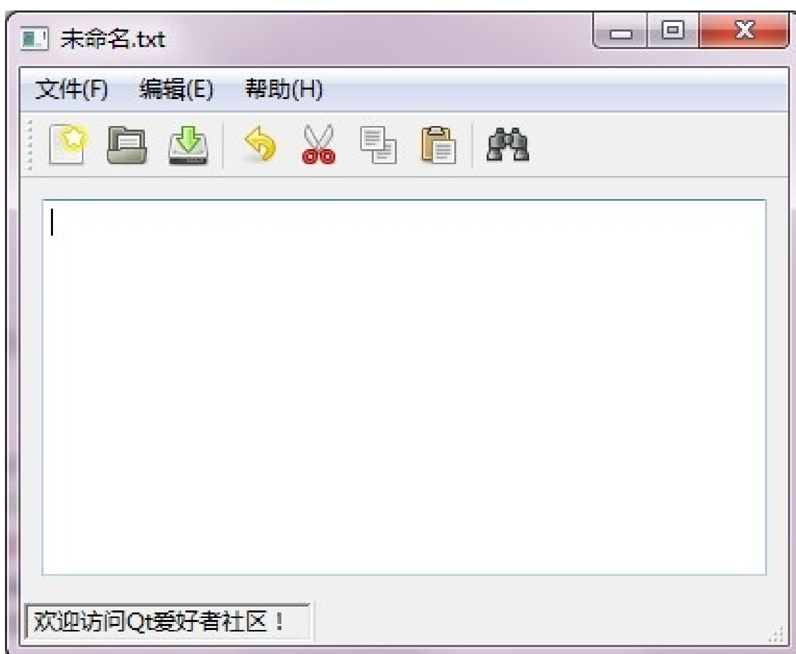
然后到构造函数中将前面添加的：

```
ui->statusBar->showMessage(tr("欢迎访问Qt爱好者社区！"), 2000);
```

一行代码注释掉，再添加如下代码：

```
statusLabel = new QLabel;
statusLabel->setMinimumSize(150, 20); // 设置标签最小大小
statusLabel->setFrameShape(QFrame::WinPanel); // 设置标签形状
statusLabel->setFrameShadow(QFrame::Sunken); // 设置标签阴影
ui->statusBar->addWidget(statusLabel);
statusLabel->setText(tr("欢迎访问Qt爱好者社区！"));
```

这时运行程序，效果如下图所示。



下面就可以在需要显示状态的时候，调用 `statusLabel` 来设置文本了。

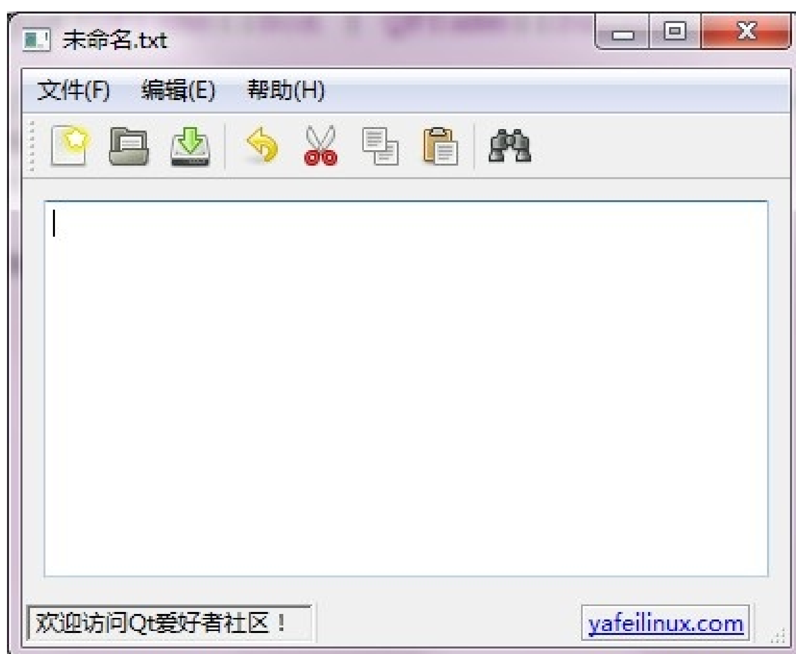
三、显示永久信息

如果要显示永久信息，要使用 `addPermanentWidget()` 函数来添加一个如 `QLabel` 一样的可以显示信息的部件，它会生成在状态栏的最右端，不会被临时消息所掩盖。

我们在构造函数中添加如下代码：

```
QLabel *permanent = new QLabel(this);
permanent->setFrameStyle(QFrame::Box | QFrame::Sunken);
permanent->setText(
  tr("<a href=\"http://www.yafeilinux.com\">yafeilinux.com</a>"));
permanent->setTextFormat(Qt::RichText);
permanent->setOpenExternalLinks(true);
ui->statusBar->addPermanentWidget(permanent);
```

这样就在状态栏的右侧添加了一个网站的超链接，点击该链接就会自动在浏览器中打开网站。运行程序，效果如下图所示。



结语

到这里整个文本编辑器的程序就算写完了。我们这里没有写帮助菜单的功能实现，大家可以自己添加。而且程序中也有很多漏洞和不完善的地方，如果有兴趣，大家也可以自己修改。因为时间和篇幅的原因，我们这里就不再过多的讲述。如果想学习一下多文档编辑器的实现，可以参考《Qt及Qt Quick开发实战精解》一书的多文档编辑器的实例。

涉及到的源码

第9篇 基础（九）Qt键盘、鼠标事件的处理

导语

事件是对各种应用程序需要知道的由应用程序内部或者外部产生的事情或者动作的通称。对于初学者，总会对Qt中信号和事件的概念混淆不清。其实，记住事件比信号更底层就可以了。比如说，我们用鼠标按下界面上的一个按钮，它会发射`clicked()`单击信号，但是，它怎么知道自己被按下的呢，那就是通过鼠标事件处理的。这里可以看到，鼠标事件比信号更底层。

在Qt中处理事件有多种方法，不过最常用的是重写Qt事件处理函数。这里我们就以鼠标事件和键盘事件为例来进行简单的介绍。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

目录

正文

一、鼠标事件

1、新建Qt Gui应用，项目名称为`myEvent`，基类更改为`QWidget`，类名为`Widget`。

2、完成项目创建后，在设计模式向界面上拖入一个`Push Button`。

3、在`widget.h`文件添加鼠标按下事件处理函数声明：

```
protected:
    void mousePressEvent(QMouseEvent *);
```

4、到`widget.cpp`文件中先添加头文件包含：

```
#include <QMouseEvent>
```

然后在下面添加函数的定义：

```
void Widget::mousePressEvent(QMouseEvent *e)
{
    ui->pushButton->setText(tr("(%1,%2)").arg(e->x()).arg(e->y()));
}
```

这里的 `arg()` 里的参数分别用来填充%1和%2处的内容， `arg()` 是 `QString` 类中的一个静态函数，使用它就可以在字符串中使用变量了。其中 `x()` 和 `y()` 分别用来返回鼠标光标所在位置的 `x` 和 `y` 坐标值。这样，当鼠标在界面上点击时，按钮就会显示出当前鼠标的坐标值。效果如下图所示。



除了鼠标按下事件，还有鼠标释放、双击、移动、滚轮等事件，其处理方式与这个例子是相似的。

二、键盘事件

1、首先在 `widget.h` 中添加 `protected` 函数声明：

```
void keyPressEvent(QKeyEvent *);
```

2、然后到 `widget.cpp` 中添加头文件包含：

```
#include <QKeyEvent>
```

3、最后添加键盘按下事件处理函数的定义：

```
void Widget::keyPressEvent(QKeyEvent *e)
{
    int x = ui->pushButton->x();
    int y = ui->pushButton->y();
    switch (e->key())
    {
    case Qt::Key_W : ui->pushButton->move(x, y-10); break;
    case Qt::Key_S : ui->pushButton->move(x, y+10); break;
    case Qt::Key_A : ui->pushButton->move(x-10, y); break;
    case Qt::Key_D : ui->pushButton->move(x+10, y); break;
    }
}
```

这里我们先获取了按钮的位置，然后使用 `key()` 函数获取按下的按键，如果是指定的W、S、A、D等按键时则移动按钮。所有的按键都在 `Qt::Key` 枚举变量中进行了定义，大家可以在帮助文档中进行查看。

结语

除了键盘按下事件，常用的还有键盘释放事件，这里就不再举例。如果想了解更多事件方面的知识，可以参考《Qt Creator快速入门》一书的第6章的内容。

涉及到的源码

第10篇 基础（十）Qt定时器和随机数

导语

在前一篇中我们介绍了键盘和鼠标事件，其实还有一个非常常用的事件，就是定时器事件，如果要对程序实现时间上的控制，那么就要使用到定时器。而随机数也是很常用的一个功能，在我们要想产生一个随机的结果时就要使用到随机数。这一篇我们就来简单介绍一下定时器和随机数。

环境是：Windows 7 + Qt 4.8.1 +Qt Creator 2.4.1

目录

正文

一、定时器

Qt中有两种方法来使用定时器，一种是定时器事件，另一种是使用信号和槽。一般使用了多个定时器时最好使用定时器事件来处理。

1．新建Qt Gui应用，项目名称为 `myTimer` ，基类选择 `QWidget` ，类名为 `Widget` 。

2．到 `widget.h` 文件中添加函数声明：

```
protected:
    void timerEvent(QTimerEvent *);
```

然后添加私有变量定义：

```
int id1, id2, id3;
```

3．下面到设计模式，向界面上拖入两个标签部件 `Label` 。

4．下面进入 `widget.cpp` 文件，先在构造函数中添加如下代码：

```
id1 = startTimer(1000);  // 开启一个1秒定时器，返回其ID
id2 = startTimer(2000);
id3 = startTimer(10000);
```

这里开启了三个定时器，分别返回了它们的 `id` ，这个 `id` 用来区分不同的定时器。定时器的时间单位是毫秒。每当一个定时器溢出时，都会调用定时器事件处理函数，我们可以在该函数中进行相应的处理。

5 · 下面添加定时器事件处理函数的定义：

```
void Widget::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == id1) {       // 判断是哪个定时器
        ui->label->setText(tr("%1").arg(qrand()%10));
    }
    else if (event->timerId() == id2) {
        ui->label_2->setText(tr("hello world!"));
    }
    else {
        qApp->quit();
    }
}
```

这里先使用 `timerId()` 函数返回了溢出的定时器的 `id` ，然后根据该 `id` 来判断是哪个定时器溢出了，并进行相应的处理。每当第一个定时器溢出时都产生一个小于10的随机数；当第二个定时器溢出时，就更改标签的文本；当第三个定时器溢出时则退出应用程序。现在可以运行程序，查看效果。

6 · 如果只是想开启少量的定时器，也可以使用信号和槽来实现。

先在 `widget.h` 中添加一个私有槽声明：

```
private slots:
    void timerUpdate();
```

然后到设计模式向界面上添加一个行编辑器部件 `Line Edit` ，再到 `widget.cpp` 中添加头文件包含：

```
#include <QTimer>
#include <QDateTime>
```

然后在构造函数中添加如下代码：

```
QTimer *timer = new QTimer(this);
//关联定时器溢出信号和相应的槽函数
connect(timer, SIGNAL(timeout()), this, SLOT(timerUpdate()));
timer->start(1000);
```

这里创建了一个定时器，并将其溢出信号和更新槽关联起来，最后使用 `start()` 函数来开启定时器。

下面添加 `timerUpdate()` 函数的定义：

```
void Widget::timerUpdate()
{
    //获取系统现在的时间
    QDateTime time = QDateTime::currentDateTime();
    //设置系统时间显示格式
    QString str = time.toString("yyyy-MM-dd hh:mm:ss dddd");
    //在标签上显示时间
    ui->lineEdit->setText(str);
}
```

这里在行编辑器中显示了当前的时间。现在可以运行程序，查看效果。

二、随机数

关于随机数，在Qt中是使用 `qrand()` 和 `qsrand()` 两个函数实现的。在前面的程序中已经看到了 `qrand()` 函数的使用，其可以产生随机数， `qrand()%10` 可以产生0-9之间的随机数。要想产生100以内的随机数就是 `%100` 。以此类推。

在使用 `qrand()` 函数产生随机数之前，一般要使用 `qsrand()` 函数为其设置初值，如果不设置初值，那么每次运行程序， `qrand()` 都会产生相同的一组随机数。为了每次运行程序时，都可以产生不同的随机数，我们要使用 `qsrand()` 设置一个不同的初值。这里使用了 `QTime` 类的 `secsTo()` 函数，它表示两个时间点之间所包含的秒数，比如代码中就是指从零点整到当前时间所经过的秒数。

下面先在 `widget.cpp` 的构造函数中添加如下代码：

```
qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
```

然后在 `timerUpdate()` 函数的最后添加如下代码：

```
int rand = qrand() % 300;            // 产生300以内的正整数
ui->lineEdit->move(rand, rand);
```

这样，每过一秒，行编辑器都会移动到一个随机的位置。大家可以运行程序，查看效果。

结语

在编程中定时器和随机数很有用，尤其是在一些需要特殊效果的程序里，比如游戏程序。如果大家想了解更多使用介绍，可以参考《Qt Creator快速入门》第6章的相关内容。

涉及到的源码下载

图形篇

第11篇 2D绘图（一）绘制简单图形

导语

Qt中提供了强大的2D绘图系统，可以使用相同的API在屏幕和绘图设备上进行绘制，它主要基于 `QPainter` 、 `QPaintDevice` 和 `QPaintEngine` 这三个类。其中 `QPainter` 用来执行绘图操作； `QPaintDevice` 提供绘图设备，它是一个二维空间的抽象，可以使用 `QPainter` 在其上进行绘制； `QPaintEngine` 提供了一些接口，可以用于 `QPainter` 在不同的设备上进行绘制。

在绘图系统中由 `QPainter` 来完成具体的绘制操作， `QPainter` 类提供了大量高度优化的函数来完成GUI编程所需要的大部分绘制工作。 `QPainter` 可以绘制一切想要的图形，从最简单的一条直线到其他任何复杂的图形，它还可以用来绘制文本和图片。 `QPainter` 可以在继承自 `QPaintDevice` 类的任何对象上进行绘制操作。

`QPainter` 一般在一个部件的重绘事件（ `Paint Event` ）的处理函数 `paintEvent()` 中进行绘制，首先要创建 `QPainter` 对象，然后进行图形的绘制，最后销毁 `QPainter` 对象。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

一、绘制一条直线

1．新建Qt Gui应用，项目名称为 `painter_1` ，类信息界面不用修改，即类名为 `MainWindow` ，基类为 `QMainWindow` 。

2．在 `mainwindow.h` 文件中添加重绘事件处理函数的声明：

```
protected:
void paintEvent(QPaintEvent *);
```

所有的绘制操作都要在这个函数里面完成。

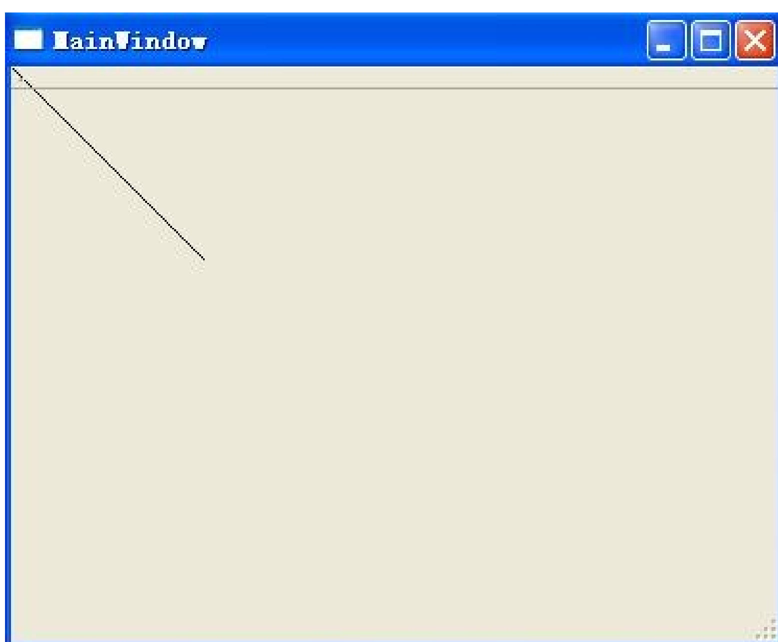
3．下面到 `mainwindow.cpp` 文件中先需要添加头文件包含：

```
#include <QPainter>
```

然后添加该函数的定义：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(QPointF(0, 0), QPointF(100, 100));
}
```

这里首先为该部件创建了一个`QPainter`对象，用于后面的绘制。然后使用`drawLine()`函数绘制了一条线段，线段的起点为`(0, 0)`，终点为`(100, 100)`，这里的单位是像素。效果如下图所示。



可以看到，在Qt窗口里面，`(0, 0)`点就是窗口的左上角，但这里是不包含外边框的。而在`MainWindow`主窗口里面绘制时，左上角并不是指中心区域的左上角，而是包含了工具栏。

4．我们将光标定位到`QPainter`类名上，然后按下键盘上的`F1`按键，这时会自动跳转到该类的帮助页面。当然，也可以到帮助模式，直接索引查找该类名。在帮助里面我们可以看到很多相关的绘制函数，如下图所示。

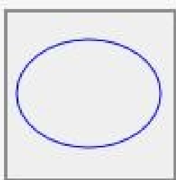
void	**drawArc** (const QRectF & *rectangle*, int *startAngle*, int *spanAngle*)
void	**drawArc** (const QRect & *rectangle*, int *startAngle*, int *spanAngle*)
void	**drawArc** (int *x*, int *y*, int *width*, int *height*, int *startAngle*, int *spanAngle*)
void	**drawChord** (const QRectF & *rectangle*, int *startAngle*, int *spanAngle*)
void	**drawChord** (const QRect & *rectangle*, int *startAngle*, int *spanAngle*)
void	**drawChord** (int *x*, int *y*, int *width*, int *height*, int *startAngle*, int *spanAngle*)
void	**drawConvexPolygon** (const QPointF * *points*, int *pointCount*)
void	**drawConvexPolygon** (const QPoint * *points*, int *pointCount*)
void	**drawConvexPolygon** (const QPolygonF & *polygon*)
void	**drawConvexPolygon** (const QPolygon & *polygon*)
void	**drawEllipse** (const QRectF & *rectangle*)

5．我们任意点击一个函数名，就会跳转到该函数的介绍段落。例如我们点击 `drawEllipse()` 函数，就跳转到了该函数的介绍处，上面还提供了一个例子。如下图所示。我们可以直接将例子里面的代码复制到 `paintEvent()` 函数里面，测试效果。

void QPainter::drawEllipse (const QRectF & *rectangle*)

Draws the ellipse defined by the given *rectangle*.

A filled ellipse has a size of *rectangle*.size(). A stroked ellipse has a size of *rectangle*.size() plus the pen width.



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);

QPainter painter(this);
painter.drawEllipse(rectangle);
```

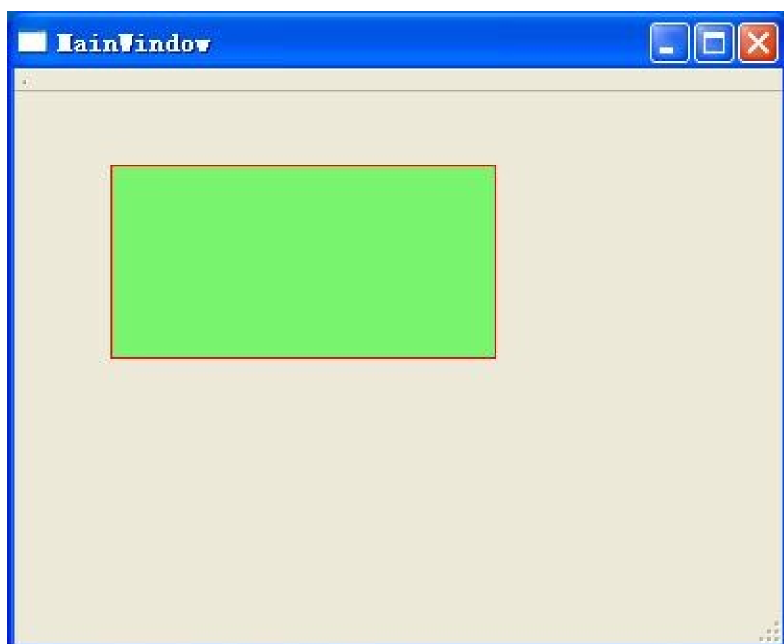
二、画笔和画刷

1．我们先将 `paintEvent( )`函数的内容更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPen pen; //画笔
    pen.setColor(QColor(255, 0, 0));
    QBrush brush(QColor(0, 255, 0, 125)); //画刷
    painter.setPen(pen); //添加画笔
    painter.setBrush(brush); //添加画刷
    painter.drawRect(50, 50, 200, 100); //绘制矩形
}
```

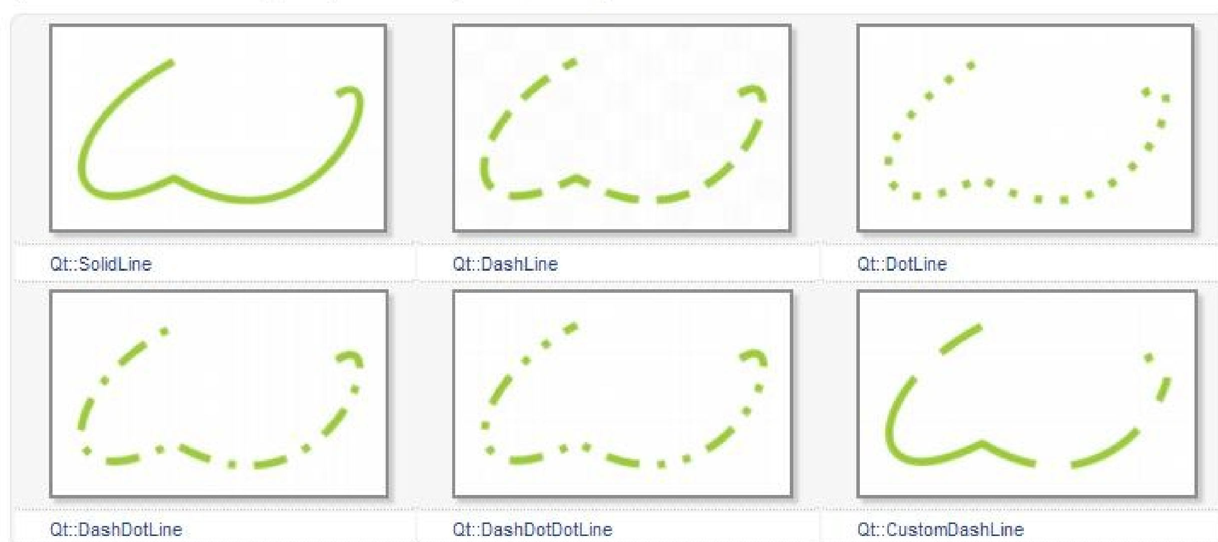
这里分别新建了一个画笔 `QPen` ，和画刷 `QBrush` 。其中画笔使用了 `setColor()` 函数为其设置了颜色，而画刷是在构建的时候直接为其设置的颜色。这里的颜色都是使用的 `QColor` 类提供的，里面如果是三个参数，那么分别是红、绿、蓝分量的值，也就是经常说的rgb，取值范围都是0-255，比如这里的 `(255, 0, 0)` 就表明红色分量为255，其他分量为0，那么出来就是红色。如果是四个参数，最后一个参数 `alpha` 是设置透明度的，取值范围也是0-255,0表示完全透明，而255表示完全不透明。

然后我们将画笔和画刷设置到了 `painter` 上，并使用 `drawRect()` 绘制了一个矩形，其左上角顶点在 `(50, 50)` ，宽为200，高为100。运行程序，效果如下图所示。

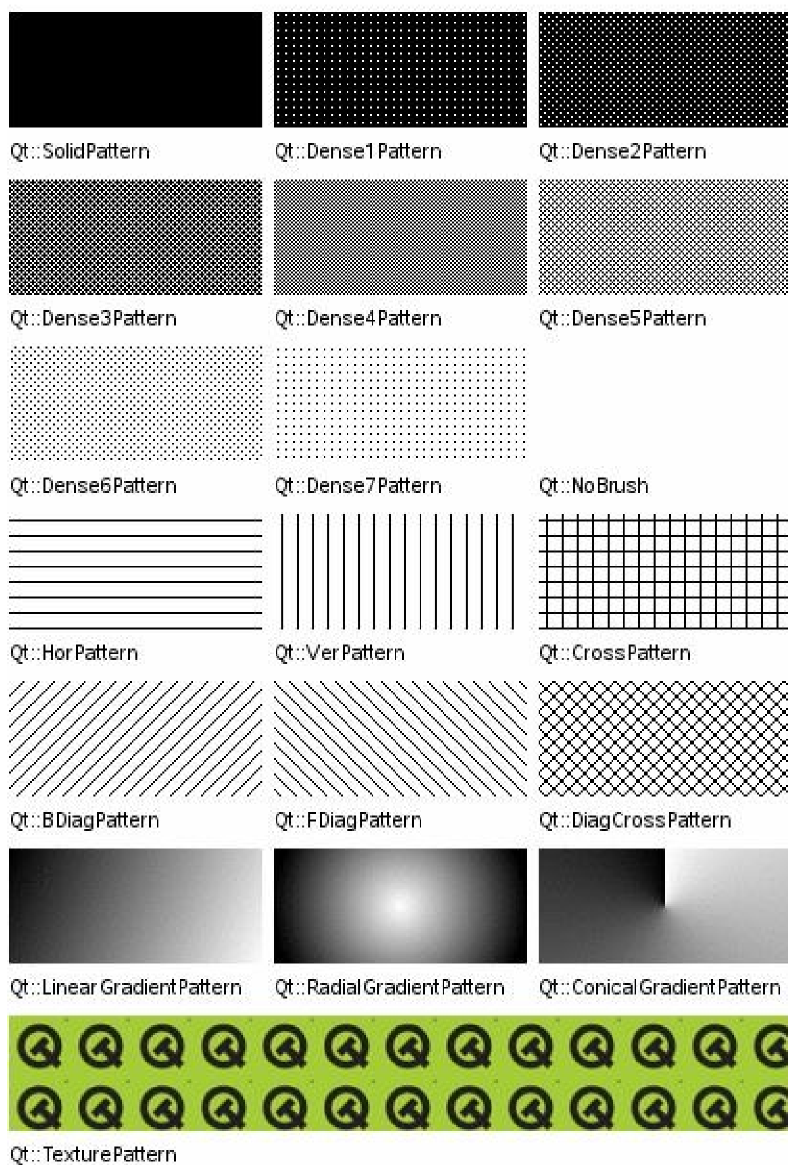


2．画笔还有许多其他的设置，可以查看该类的帮助文档。例如，可以使用 `pen.setStyle()` 来设置画笔样式，可用的画笔样式如下图所示。

Qt provides several built-in styles represented by the Qt::PenStyle enum:



3．画刷也有很多其他设置，这个也可以查看其帮助文档。在Qt中为画刷提供了一些可用的样式，可以使用 `setStyle()` 函数来设置。如下图所示。



这里面包含了渐变填充效果，这个会在下一节讲到。

4、下面我们写一个简单的例子演示一下。将`paintEvent()`函数更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPen pen(Qt::DotLine);
    QBrush brush(Qt::blue);
    brush.setStyle(Qt::HorPattern);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.drawRect(50,50,200,200);
}
```

这里的颜色使用了Qt预定义的颜色，可以在帮助中索引`Qt::GlobalColor`关键字查看。如下图所示。

enum Qt::GlobalColor

Qt's predefined QColor objects:

Constant	Value	Description
Qt::white	3	White (#ffffff)
Qt::black	2	Black (#000000)
Qt::red	7	Red (#ff0000)
Qt::darkRed	13	Dark red (#800000)
Qt::green	8	Green (#00ff00)
Qt::darkGreen	14	Dark green (#008000)
Qt::blue	9	Blue (#0000ff)
Qt::darkBlue	15	Dark blue (#000080)
Qt::cyan	10	Cyan (#00ffff)
Qt::darkCyan	16	Dark cyan (#008080)
Qt::magenta	11	Magenta (#ff00ff)
Qt::darkMagenta	17	Dark magenta (#800080)
Qt::yellow	12	Yellow (#ffff00)
Qt::darkYellow	18	Dark yellow (#808000)
Qt::gray	5	Gray (#a0a0a4)
Qt::darkGray	4	Dark gray (#808080)
Qt::lightGray	6	Light gray (#c0c0c0)
Qt::transparent	19	a transparent black value (i.e., QColor(0, 0, 0, 0))
Qt::color0	0	0 pixel value (for bitmaps)
Qt::color1	1	1 pixel value (for bitmaps)

现在运行程序，效果如下图所示。

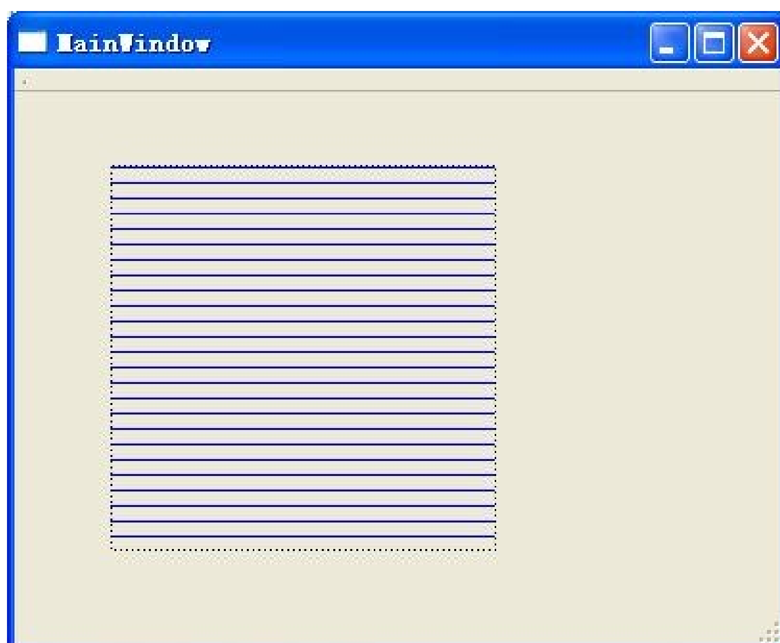
enum Qt::GlobalColor

Qt's predefined QColor objects:

Constant	Value	Description
Qt::white	3	White (#ffffff)
Qt::black	2	Black (#000000)
Qt::red	7	Red (#ff0000)
Qt::darkRed	13	Dark red (#800000)
Qt::green	8	Green (#00ff00)
Qt::darkGreen	14	Dark green (#008000)
Qt::blue	9	Blue (#0000ff)
Qt::darkBlue	15	Dark blue (#000080)
Qt::cyan	10	Cyan (#00ffff)
Qt::darkCyan	16	Dark cyan (#008080)
Qt::magenta	11	Magenta (#ff00ff)
Qt::darkMagenta	17	Dark magenta (#800080)
Qt::yellow	12	Yellow (#ffff00)
Qt::darkYellow	18	Dark yellow (#808000)
Qt::gray	5	Gray (#a0a0a4)
Qt::darkGray	4	Dark gray (#808080)
Qt::lightGray	6	Light gray (#c0c0c0)
Qt::transparent	19	a transparent black value (i.e., QColor(0, 0, 0, 0))
Qt::color0	0	0 pixel value (for bitmaps)
Qt::color1	1	1 pixel value (for bitmaps)

三、绘制弧线

为了帮助大家学习，这里再举一个绘制弧线的例子，其实在帮助文档中已经给出了这个例子。如下图所示。



我们将 `paintEvent()` 函数更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QRectF rectangle(10.0, 20.0, 80.0, 60.0); //矩形
    int startAngle = 30 * 16;     //起始角度
    int spanAngle = 120 * 16;   //跨越度数
    QPainter painter(this);
    painter.drawArc(rectangle, startAngle, spanAngle);
}
```

这里要说明的是，画弧线时，角度被分成了十六分之一，就是说，要想为30度，就得是 `30*16` 。它有起始角度和跨度，还有位置矩形，要想画出自己想要的弧线，就要有一定的几何知识了。这里就不再祥述。

结语

这一节我们只是简单介绍了一下怎么使用 `QPainter` 在窗口界面上进行绘制，还涉及到了画笔、画刷，以及使用帮助文档的一些内容。如果要更加系统、详细的学习这些基础知识，可以查看《Qt Creator快速入门》的第10章。

涉及到的源码下载

第12篇 2D绘图（二）渐变填充

导语

在前一节提到了在画刷中可以使用渐变填充。`QGradient`类就是用来和`QBrush`一起指定渐变填充的。Qt现在支持三种类型的渐变填充：

- 线性渐变（linear gradient）在开始点和结束点之间插入颜色；
- 辐射渐变（radial gradient）在焦点和环绕它的圆环间插入颜色；
- 锥形渐变（Conical）在圆心周围插入颜色。

这三种渐变分别由`QGradient`的三个子类来表示，`QLinearGradient`表示线性渐变，`QRadialGradient`表示辐射渐变，`QConicalGradient`表示锥形渐变。

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

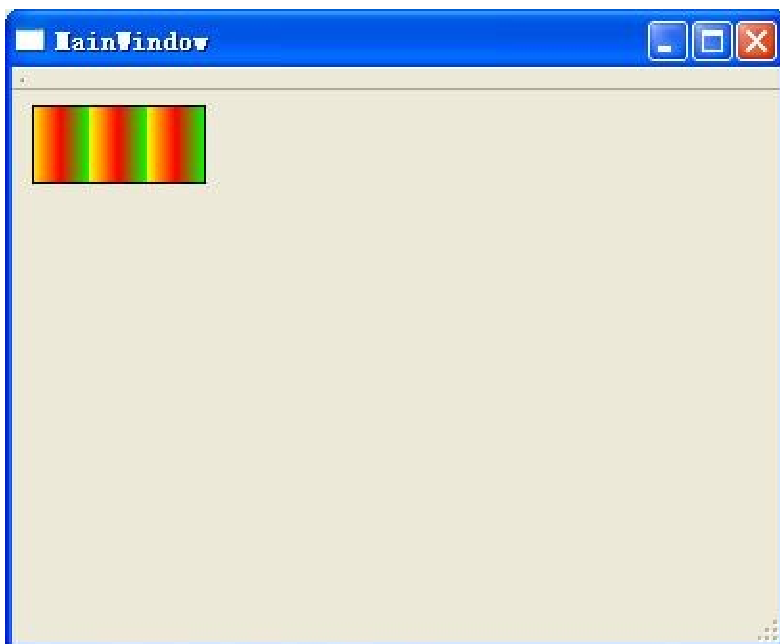
正文

一、线性渐变

1．我们仍然在上一节创建的项目中进行讲解。更改`paintEvent()`函数如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
     //线性渐变
    QLinearGradient linearGradient(QPointF(40, 190), QPointF(70, 190));
    //插入颜色
    linearGradient.setColorAt(0, Qt::yellow);
    linearGradient.setColorAt(0.5, Qt::red);
    linearGradient.setColorAt(1, Qt::green);
    //指定渐变区域以外的区域的扩散方式
    linearGradient.setSpread(QGradient::RepeatSpread);
    //使用渐变作为画刷
    QPainter painter(this);
    painter.setBrush(linearGradient);
    painter.drawRect(10, 20, 90, 40);
}
```

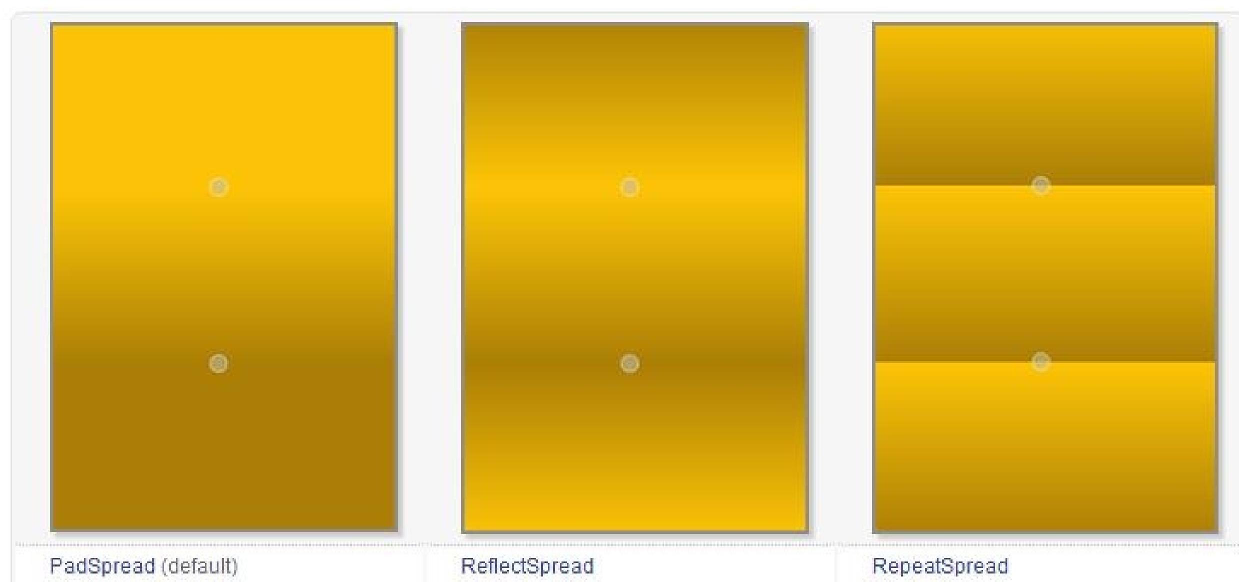
运行程序，效果如下图所示。



2、介绍

对于线性渐变 `QLinearGradient::QLinearGradient ( const QPointF & start, constQPointF & finalStop )` 需要指定开始点 `start` 和结束点 `finalStop` ，然后将开始点和结束点之间的区域进行等分，开始点的位置为 `0.0` ，结束点的位置为 `1.0` ，而它们之间的位置按照距离比例进行设定，然后使用 `QGradient::setColorAt( qreal position, const QColor & color )` 函数在指定的位置 `position` 插入指定的颜色 `color` ，当然，这里的 `position` 的值要在0到1之间。

这里还可以使用 `setSpread()` 函数来设置填充的扩散方式，即指明在指定区域以外的区域怎样进行填充。扩散方式由 `QGradient::Spread` 枚举变量定义，它一共有三个值，分别是 `QGradient::PadSpread` ，使用最接近的颜色进行填充，这是默认值，如果我们不使用 `setSpread()` 指定扩散方式，那么就会默认使用这种方式； `QGradient::RepeatSpread` ，在渐变区域以外的区域重复渐变； `QGradient::ReflectSpread` ，在渐变区域以外将反射渐变。在线性渐变中这三种扩散方式的效果下图所示。要使用渐变填充，可以直接在 `setBrush()` 中使用，这时画刷风格会自动设置为相对应的渐变填充。

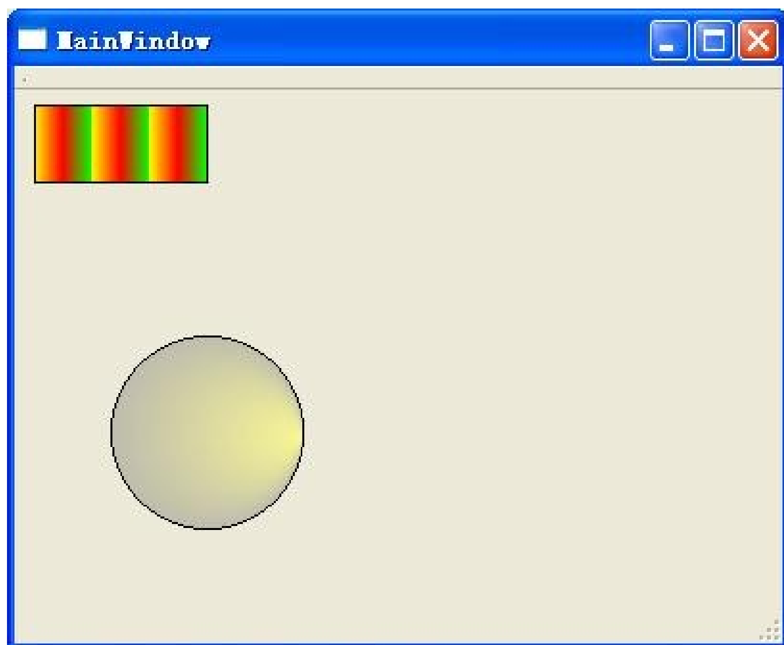


二、辐射渐变

1．继续在 `paintEvent()` 函数中添加如下代码：

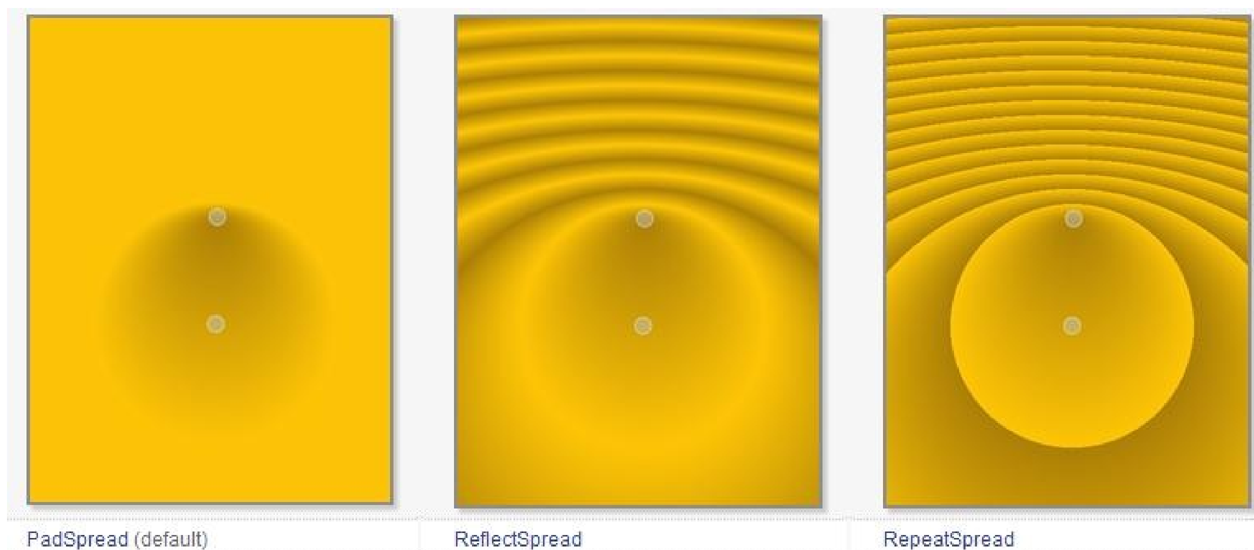
```
//辐射渐变
QRadialGradient radialGradient(QPointF(100, 190),50,QPointF(275,200));
radialGradient.setColorAt(0, QColor(255, 255, 100, 150));
radialGradient.setColorAt(1, QColor(0, 0, 0, 50));
painter.setBrush(radialGradient);
painter.drawEllipse(QPointF(100, 190), 50, 50);
```

运行程序，效果如下图所示。



2．介绍

对于辐射渐变 `QRadialGradient::QRadialGradient( const QPointF & center, qreal radius, const QPointF &` 需要指定圆心 `center` 和半径 `radius` ，这样就确定了一个圆，然后再指定一个焦点 `focalPoint` 。焦点的位置为0，圆环的位置为1，然后在焦点和圆环间插入颜色。辐射渐变也可以使用 `setSpread()` 函数设置渐变区域以外的区域的扩散方式，三种扩散方式的效果如下图所示。



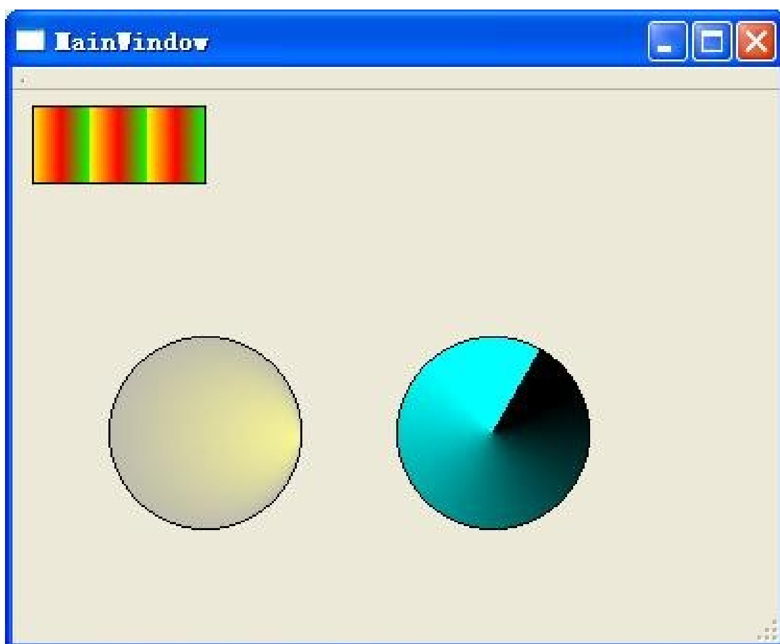
PadSpread (default) ReflectSpread RepeatSpread

三、锥形渐变

1. 接着在 `paintEvent()` 函数里面添加如下代码：

```
//锥形渐变
QConicalGradient conicalGradient(QPointF(250, 190), 60);
conicalGradient.setColorAt(0.2, Qt::cyan);
conicalGradient.setColorAt(0.9, Qt::black);
painter.setBrush(conicalGradient);
painter.drawEllipse(QPointF(250, 190), 50, 50);
```

运行程序，效果如下图所示。



2・介绍

对于锥形渐变 `QConicalGradient::QConicalGradient ( const QPointF & center,qreal angle )` 需要指定中心点 `center` 和一个角度 `angle` （其值在0到360之间），然后沿逆时针从给定的角度开始环绕中心点插入颜色。这里给定的角度沿逆时针方向开始的位置为0，旋转一圈后为1。`setSpread()` 函数对于锥形渐变没有效果。

结语

本节在前面的基础上，简单介绍了一下常用的三种渐变填充。如果大家可以熟练使用这几种填充效果，那么就可以实现非常漂亮的界面。另外，还可以给画笔设置渐变颜色，这样就可以绘制出特殊效果的线条和文字，这个可以参考《Qt Creator快速入门》的相关内容。

涉及到的相关源码

第13篇 2D绘图（三）绘制文字

导语

Qt中除了绘制图形以外，还可以使用 `QPainter::darwText()` 函数来绘制文字，也可以使用 `QPainter::setFont()` 设置文字所使用的字体，使用 `QPainter::fontInfo()` 函数可以获取字体的信息，它返回 `QFontInfo` 类对象。在绘制文字时会默认使用抗锯齿。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

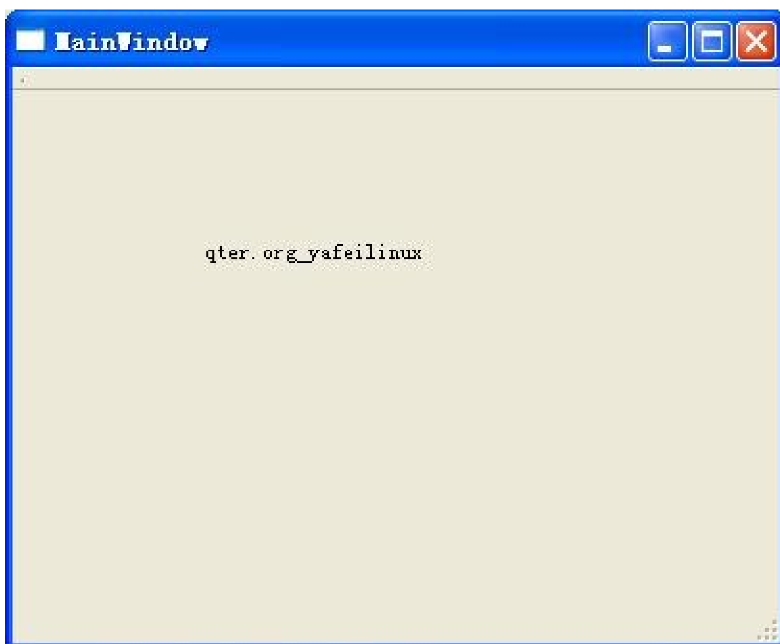
正文

一、基本绘制

我们接着在上一节的项目上进行讲解，首先将 `paintEvent()` 函数更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawText(100, 100, "qter.org_yafeilinux");
}
```

这样就在 `(100, 100)` 的位置绘制了一个字符串。效果如下图所示。



二、控制文字的位置

1．我们先到QPainter的帮助文档页面，然后查看 `drawText()` 函数的重载形式，找到： `QPainter::drawText ( const QRectF &rectangle, int flags, const QString & text, QRectF * bo` ，如下图所示。

void **QPainter::drawText** (**const QRectF & *rectangle***, int ***flags***, **const QString & *text***, **QRectF * *boundingRect* = 0**)

This is an overloaded function.

Draws the given *text* within the provided *rectangle*.



它的第一个参数指定了绘制文字所在的矩形；第二个参数指定了文字在矩形中的对齐方式，它由 `Qt::AlignmentFlag` 枚举变量进行定义，不同对齐方式也可以使用 `|` 操作符同时使用，这里还可以使用 `Qt::TextFlag` 定义的其他一些标志，比如自动换行等；第三个参数就是所要绘制的文字，这里可以使用 `\n` 来实现换行；第四个参数一般不用设置。

2．下面我们来看一个例子。为了更明显的看到文字在指定矩形中的位置，我们绘制出这个矩形。将 `paintEvent()` 函数更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    //设置一个矩形
    QRectF rect(20, 20, 300, 200);
    //为了更直观地看到字体的位置，我们绘制出这个矩形
    painter.drawRect(rect);
    painter.setPen(QColor(Qt::red));
    //我们这里先让字体水平居中
    painter.drawText(rect, Qt::AlignHCenter, "yafeilinux");
}
```

现在运行程序，效果如下图所示。



可用的对齐方式如下图所示。

- Qt::AlignLeft
- Qt::AlignRight
- Qt::AlignHCenter
- Qt::AlignJustify
- Qt::AlignTop
- Qt::AlignBottom
- Qt::AlignVCenter
- Qt::AlignCenter
- Qt::TextDontClip
- Qt::TextSingleLine
- Qt::TextExpandTabs
- Qt::TextShowMnemonic
- Qt::TextWordWrap
- Qt::TextIncludeTrailingSpaces

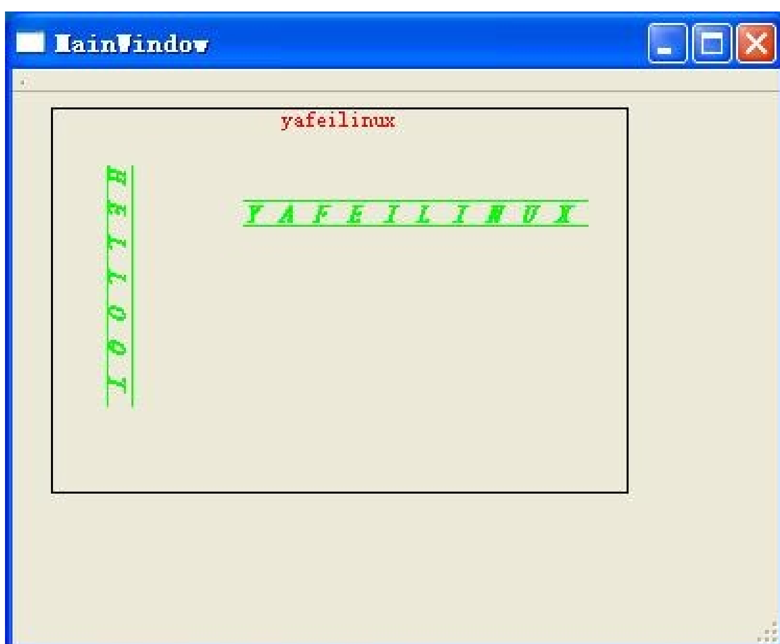
三、使用字体

为了绘制漂亮的文字，可以使用 `QFont` 类来设置文字字体。大家也可以先在帮助文档中查看该类的介绍。下面将最常用的一些设置进行演示。

在 `paintEvent()` 函数中继续添加如下代码：

```
QFont font("宋体", 15, QFont::Bold, true);
//设置下划线
font.setUnderline(true);
//设置上划线
font.setOverline(true);
//设置字母大小写
font.setCapitalization(QFont::SmallCaps);
//设置字符间的间距
font.setLetterSpacing(QFont::AbsoluteSpacing, 10);
//使用字体
painter.setFont(font);
painter.setPen(Qt::green);
painter.drawText(120, 80, tr("yafeilinux"));
painter.translate(50, 50);
painter.rotate(90);
painter.drawText(0, 0, tr("helloqt"));
```

这里创建了 `QFont` 字体对象，使用的构造函数为 `QFont::QFont ( const QString & family,int pointSize = -1, int weight = -1, bool italic =` ，第一个参数设置字体的 `family` 属性，这里使用的字体族为宋体，可以使用 `QFontDatabase` 类来获取所支持的所有字体；第二个参数是点大小，默认大小为12；第三个参数为 `weight` 属性，这里使用了粗体；最后一个属性设置是否使用斜体。然后我们又使用了其他几个函数来设置字体的格式，最后调用 `setFont()` 函数来使用该字体，并使用 `drawText()` 函数的另一种重载形式在点 `（120, 80）` 绘制了文字。后面又将坐标系统平移并旋转，然后再次绘制了文字。运行程序，效果如下图所示。



结语

这一节最后的例子中使用了`rotate()`函数来旋转坐标系统，从而绘制出了纵向的文字。这个将会在后面的篇章中介绍到。

涉及到的源码下载

第14篇 2D绘图（四）绘制路径

导语

如果要绘制一个复杂的图形，尤其是要重复绘制这样的图形，那么可以使用 `QPainterPath` 类，然后使用 `QPainter::drawPath()` 来进行绘制。 `QPainterPath` 类为绘制操作提供了一个容器，可以用来创建图形并且重复使用。一个绘图路径就是由多个矩形、椭圆、线条或者曲线等组成的对象，一个路径可以是封闭的，例如矩形和椭圆；也可以是非封闭的，例如线条和曲线。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

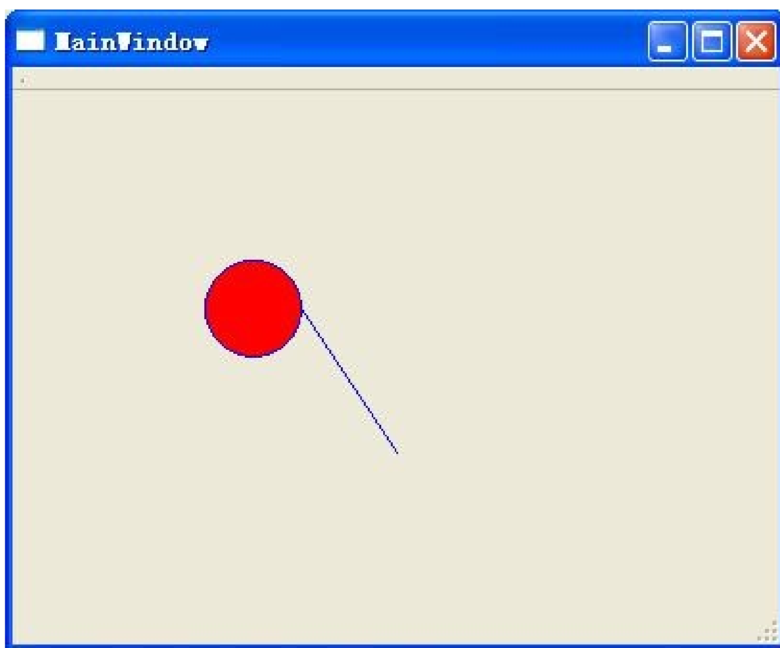
正文

一、简单的使用路径

依然在前面的项目中进行讲解。更改 `paintEvent()` 函数如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.addEllipse(100, 100, 50, 50);
    path.lineTo(200, 200);
    QPainter painter(this);
    painter.setPen(Qt::blue);
    painter.setBrush(Qt::red);
    painter.drawPath(path);
}
```

当创建一个 `QPainterPath` 对象后，可以使用 `lineTo()` 、 `arcTo()` 、 `cubicTo()` 和 `quadTo()` 等函数将直线或者曲线添加到路径中。运行程序，效果如下图所示。

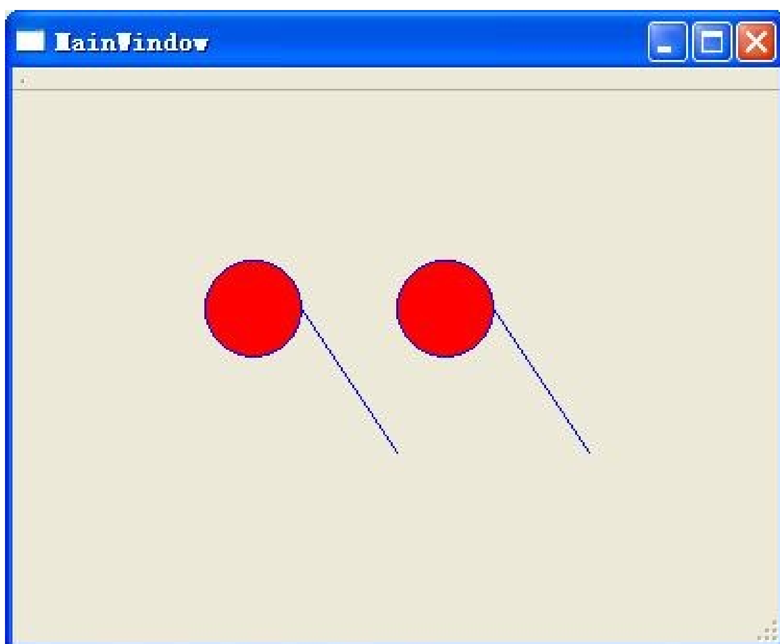


二、复制图形

如果只是简单的将几个图形拼接在一起，其实完全没有必要用路径，之所以要引入路径，就是因为它的一个非常有用的功能：复制图形路径。我们在 `painEvent()` 函数中继续添加下面几行代码：

```
QPainterPath path2;
path2.addPath(path);
path2.translate(100,0);
painter.drawPath(path2);
```

现在运行程序，效果如下图所示。



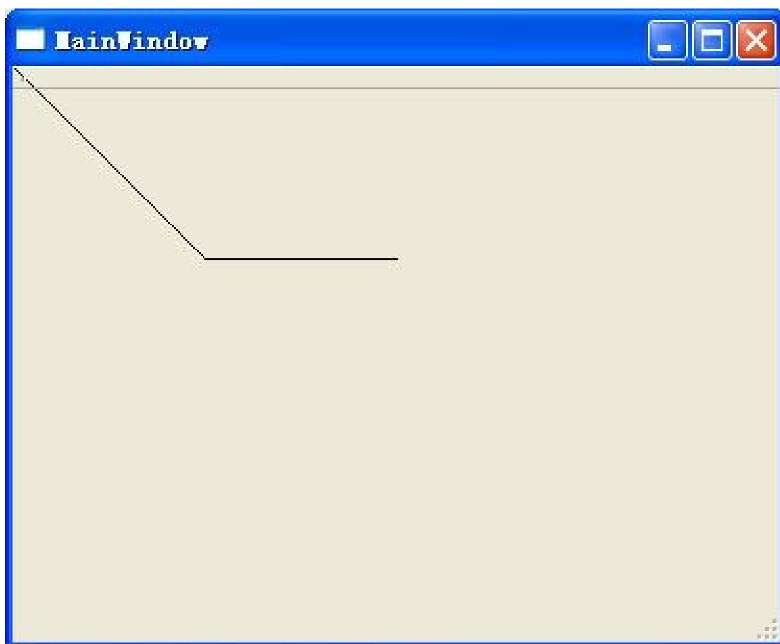
可以看到，对于已经绘制好的路径，可以非常简单的进行重复绘制。

三、绘制图形时的当前位置

1. 我们先来看个例子，将 `paintEvent()` 函数更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.lineTo(100,100);
    path.lineTo(200,100);
    QPainter painter(this);
    painter.drawPath(path);
}
```

程序运行效果如下图所示。

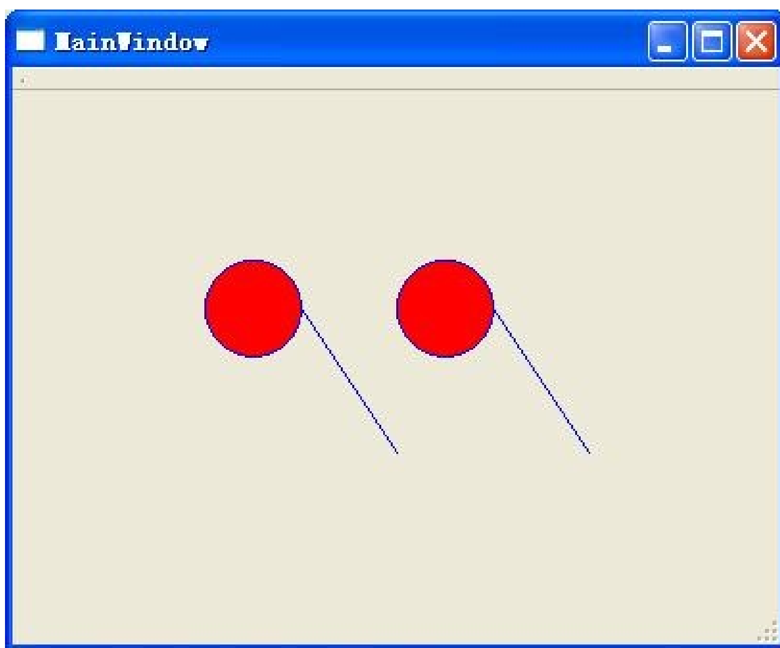


可以看到，创建路径后，默认是从 `(0, 0)` 点开始绘制的，当绘制完第一条直线后当前位置是 `(100, 100)` 点，从这里开始绘制第二条直线。绘制完第二条直线后，当前位置是 `(200,100)` 。

2. 再来看一个例子。将 `paintEvent()` 函数的内容更改如下：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.addRect(50, 50, 40, 40);
    path.lineTo(200, 200);
    QPainter painter(this);
    painter.drawPath(path);
}
```

运行程序，效果如下图所示。

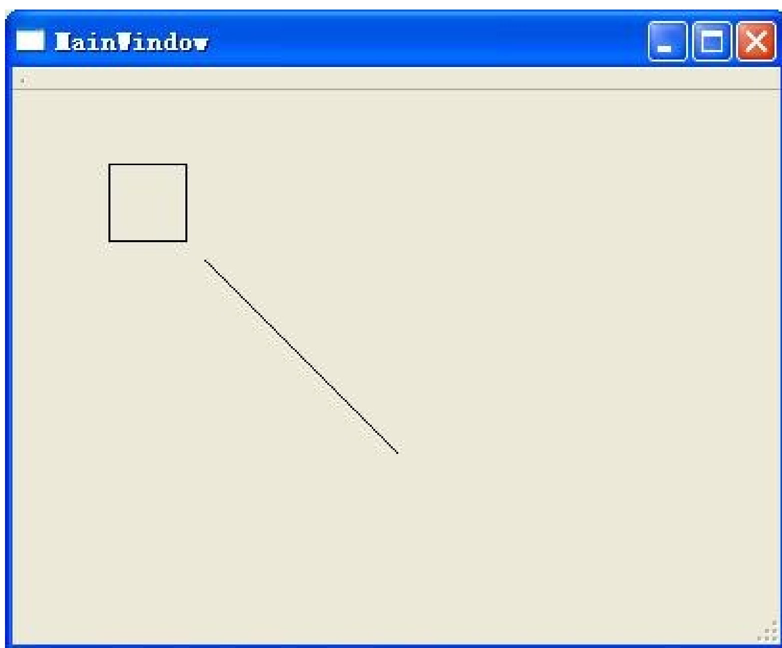


可以发现，当绘制完矩形后，当前位置在矩形的左上角顶点，然后从这里开始绘制后面的直线。

4．我们也可以使用 `moveTo()` 函数来改变当前点的位置。例如将上面的代码更改为：

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainterPath path;
    path.addRect(50, 50, 40, 40);
    //移动到(100, 100)点
    path.moveTo(100, 100);
    path.lineTo(200, 200);
    QPainter painter(this);
    painter.drawPath(path);
}
```

这样当绘制完矩形以后，就会移动到 `(100, 100)` 点进行后面的绘制。程序运行效果如下图所示。



结语

这里只讲解了 `QPainterPath` 最基本的应用，使用好这个类可以绘制出很多特效图形。如果绘制的两个图形有交集，那么还要涉及到相交部分的填充规则问题，这部分内容可以参考《Qt Creator快速入门》第10章的相关内容。

涉及到的源码下载

第15篇 2D绘图（五）绘制图片

导语

Qt提供了四个类来处理图像数据：`QImage`、`QPixmap`、`QBitmap`和`QPicture`，它们也都是常用的绘图设备。其中`QImage`主要用来进行I/O处理，它对I/O处理操作进行了优化，而且也可以用来直接访问和操作像素；`QPixmap`主要用来在屏幕上显示图像，它对在屏幕上显示图像进行了优化；`QBitmap`是`QPixmap`的子类，它是一个便捷类，用来处理颜色深度为1的图像，即只能显示黑白两种颜色；`QPicture`用来记录并重演`QPainter`命令。这一节我们只讲解`QPixmap`。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

一、简单绘制图片

1．这次我们重新创建一个Qt Gui应用，项目名称为`painter_2`，在类信息页面，将基类更改为`QDialog`，类名使用默认的`Dialog`即可。

2．然后在源码目录中复制一张图片，比如这里是一张背景透明的`logo.png`图片，如下图所示。



3、在 `dialog.h` 文件中添加重绘事件处理函数的声明：

```
protected:
    void paintEvent(QPaintEvent *);
```

4、到 `dialog.cpp` 文件中先添加头文件包含 `#include <QPainter>` ，然后添加函数的定义：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load("../painter_2/logo.png");
    painter.drawPixmap(0, 0, 129, 66, pix);
}
```

这里使用了相对路径，因为Qt Creator默认是使用影子构建，即编译生成的文件在 `painter_2-build-desktop-Debug` 这样的目录里面，而这个目录就是当前目录，所以源码目录就是其上级目录了。大家可以根据自己的实际情况来更改路径，也可以使用绝对路径，不过最好使用资源文件来存放图片。 `drawPixmap()` 函数在给定的矩形中来绘制图片，这里矩形的左上角顶点为 `(0, 0)` 点，宽129，高66，如果这个跟图片的大小不相同，默认会拉伸图片。运行效果如下图所示。



（注意：下面的操作涉及到了坐标系统，这里不再详细讲解，大家先进行操作查看效果，具体的坐标内容将在下一节讲解。）

二、平移图片

`QPainter` 类中的 `translate()` 函数实现坐标原点的改变，改变原点后，此点将会成为新的原点 `（0，0）` 。下面来看一个例子。

在 `paintEvent()` 函数里面继续添加如下代码：

```
painter.translate(100, 100); //将（100，100）设为坐标原点
painter.drawPixmap(0, 0, 129, 66, pix);
```

运行程序，效果如下图所示。



这里将 `（100，100）` 设置为了新的坐标原点，所以下面在 `（0，0）` 点贴图，就相当于在以前的 `（100，100）` 点贴图。

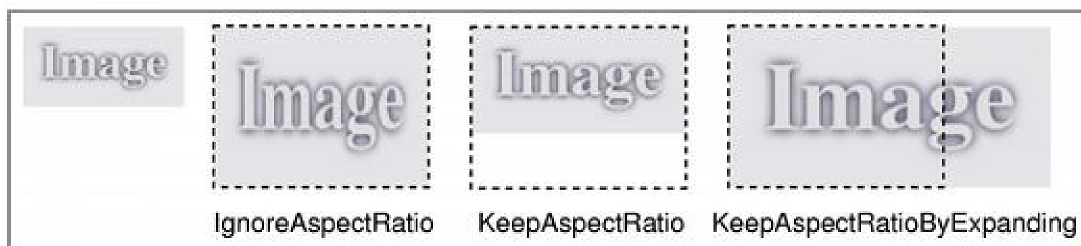
三、缩放图片

我们可以使用 `QPixmap` 类中的 `scaled()` 函数来实现图片的放大和缩小。

在 `paintEvent()` 函数中继续添加代码：

```
qreal width = pix.width(); //获得以前图片的宽和高
qreal height = pix.height();
//将图片的宽和高都扩大两倍，并且在给定的矩形内保持宽高的比值不变
pix = pix.scaled(width*2, height*2, Qt::KeepAspectRatio);
painter.drawPixmap(70, 70,pix);
```

其中参数 `Qt::KeepAspectRatio` ，是图片缩放的方式。我们可以查看其帮助。将鼠标指针放到该代码上，当出现F1提示时，按下F1键，这时就可以查看其帮助了。当然我们也可以直接在帮助里查找该关键字。如下图所示。



这里有三个值，只看其图片就可大致明白， `Qt::IgnoreAspectRatio` 是不保持图片的宽高比， `Qt::KeepAspectRatio` 是在给定的矩形中保持宽高比，最后一个也是保持宽高比，但可能超出给定的矩形。这里给定的矩形是由我们显示图片时给定的参数决定的，例如 `painter.drawPixmap(0,0,100,100,pix);` 就是在以 `（0，0）` 点为起始点的宽和高都是100的矩形中。

运行程序效果如下图所示。



四、旋转图片

旋转使用的是 `QPainter` 类的 `rotate()` 函数，它默认是以原点为中心进行旋转的。我们要改变旋转的中心，可以使用前面讲到的 `translate()` 函数完成。

在 `paintEvent()` 函数中继续添加如下代码：

```
painter.translate(64, 33); //让图片的中心作为旋转的中心
painter.rotate(90); //顺时针旋转90度
painter.translate(-64,-33); //使原点复原
painter.drawPixmap(100, 100, 129, 66, pix);
```

这里必须先改变旋转中心，然后再旋转，然后再将原点复原，才能达到想要的效果。 运行程序，效果如图所示。



五、扭曲图片

实现图片的扭曲，是使用的`QPainter`类的`shear(qreal sh，qreal sv)`函数完成的。它有两个参数，前面的参数实现横行变形，后面的参数实现纵向变形。当它们的值为0时，表示不扭曲。

在`paintEvent()`中继续添加如下代码：

```
painter.shear(0.5, 0); //横向扭曲
painter.drawPixmap(100, 0, 129, 66, pix);
```

运行效果如下图所示。



结语

这一节中只讲解了`QPixmap`类的简单应用。对于后面讲到的图片变形的应用，细心的读者可能已经发现了，旋转了坐标系统以后再绘制图片都是纵向的，这就是因为旋转了坐标系统而没有进行恢复造成的。具体的坐标操作我们会在下一节讲解。

如果大家还想系统的学习其他绘图类的应用，可以参考《Qt Creator快速入门》第10章的相关内容。

第16篇 2D绘图（六）坐标系统

导语

前面一节我们讲解了图片的显示，其中很多地方都用到了坐标的变化。这一节我们将讲解Qt的坐标系统，分为两部分来讲解：第一部分主要讲解前面一节的那几个函数，它们分别是`translate()`平移变换、`scale()`比例变换、`rotate()`旋转变换、`shear()`扭曲变换。最后还会介绍两个有用的函数`save()`和`restore()`，利用它们来保存和弹出坐标系的状态，从而实现快速利用几个变换函数来绘图。

第二部分会和大家一起来研究一下Qt的坐标系统，其中可能会涉及到多个坐标，大家一定要亲自动手操作感悟一下，不然很难理解的！

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

正文

第一部分 Qt坐标系统应用

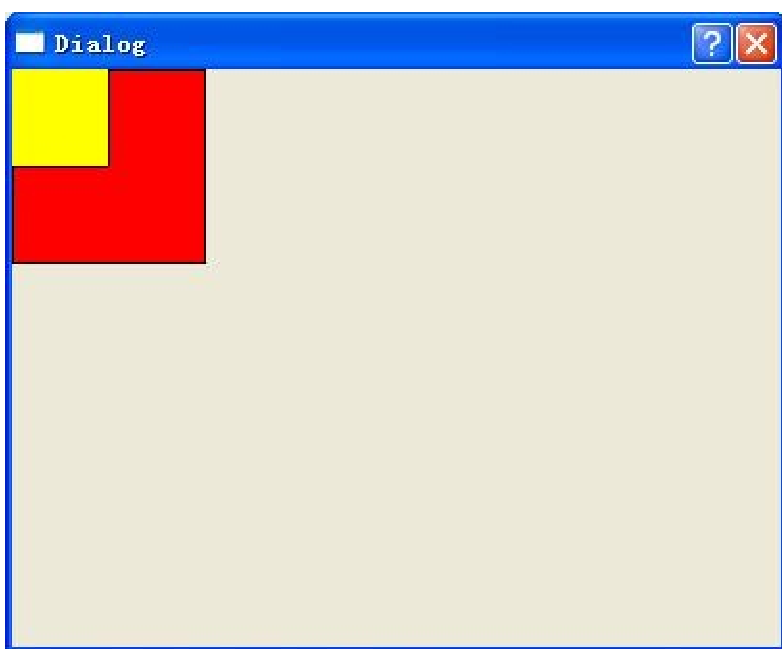
一、坐标系统简介

Qt中每一个窗口都有一个坐标系统，默认的，窗口左上角为坐标原点，水平向右依次增大，水平向左依次减小，垂直向下依次增大，垂直向上依次减小。原点即为`（0，0）`点，以像素为单位增减。

下面仍然在上一节的程序中进行代码演示，更改`paintEvent()`的内容如下：

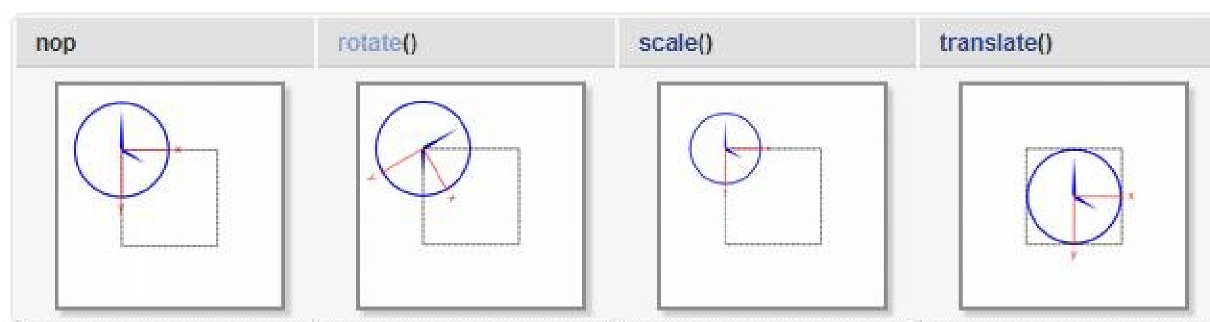
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setBrush(Qt::red);
    painter.drawRect(0, 0, 100, 100);
    painter.setBrush(Qt::yellow);
    painter.drawRect(-50, -50, 100, 100);
}
```

我们先在原点 `（0，0）` 绘制了一个长宽都是100像素的红色矩形，又在 `（-50，-50）` 点绘制了一个同样大小的黄 色矩形。可以看到，我们只能看到黄 色矩形的四分之一部分。运行程序，效果如下图所示。



二、坐标系统变换

默认的， `QPainter` 在相关设备的坐标系统上进行绘制，在进行绘图时，可以使用 `QPainter::scale()` 函数缩放坐标系统；使用 `QPainter::rotate()` 函数顺时针旋转坐标系统；使用 `QPainter::translate()` 函数平移坐标系统；还可以使用 `QPainter::shear()` 围绕原点来扭曲坐标系统。如下图所示。

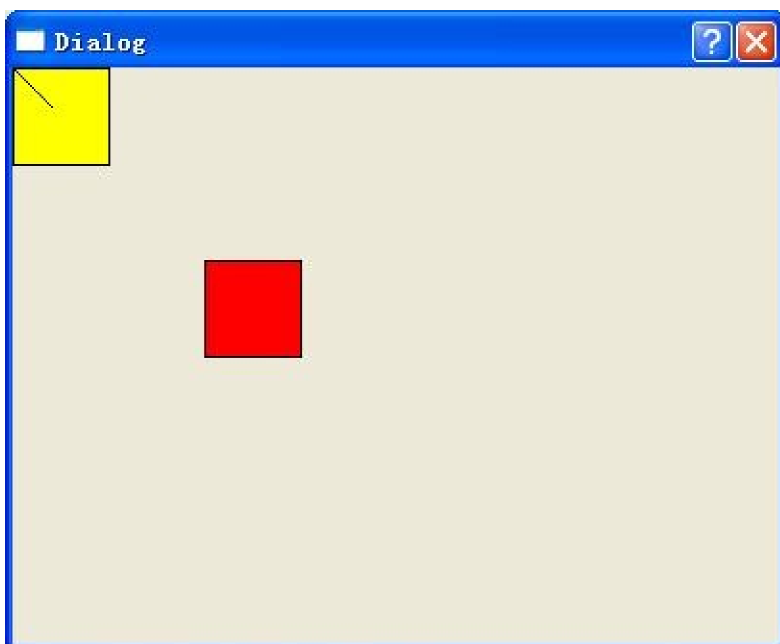


坐标系统的2D变换由 `QTransform` 类实现，我们可以使用前面提到的那些便捷函数进行坐标系统变换，当然也可以通过 `QTransform` 类实现。

1．平移变换。将 `paintEvent()` 函数内容更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 平移变换
    QPainter painter(this);
    painter.setBrush(Qt::yellow);
    painter.drawRect(0, 0, 50, 50);
    painter.translate(100, 100);  //将点（100，100）设为原点
    painter.setBrush(Qt::red);
    painter.drawRect(0, 0, 50, 50);
    painter.translate(-100, -100);
    painter.drawLine(0, 0, 20, 20);
}
```

运行程序，效果如下图所示。

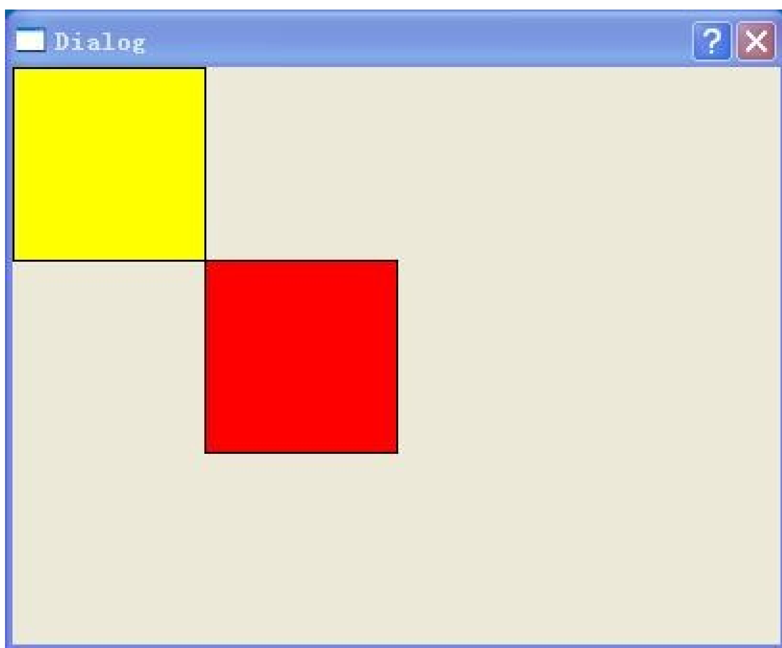


这里先在原点 `(0, 0)` 绘制了一个宽、高均为50的正方形，然后使用 `translate()` 函数将坐标系统进行了平移，使 `(100, 100)` 点成为了新原点，所以我们再次进行绘制的时候，虽然 `drawRect()` 中的逻辑坐标还是 `(0, 0)` 点，但实际显示出来的却是在 `(100, 100)` 点的红色正方形。可以再次使用 `translate()` 函数进行反向平移，使原点重新回到窗口左上角。

2．缩放变换。将 `paintEvent()` 函数中的内容更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 缩放
    QPainter painter(this);
    painter.setBrush(Qt::yellow);
    painter.drawRect(0, 0, 100, 100);
    painter.scale(2, 2); //放大两倍
    painter.setBrush(Qt::red);
    painter.drawRect(50, 50, 50, 50);
}
```

运行程序，效果如下图所示。



可以看到，当我们使用 `scale()` 函数将坐标系统的横、纵坐标都放大两倍以后，逻辑上的 `（50，50）` 点变成了窗口上的 `（100，100）` 点，而逻辑上的长度50，绘制到窗口上的长度却是100。

3．扭曲变换。将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
            // 扭曲
            QPainter painter(this);
            painter.setBrush(Qt::yellow);
            painter.drawRect(0, 0, 50, 50);
            painter.shear(0, 1); //纵向扭曲变形
            painter.setBrush(Qt::red);
            painter.drawRect(50, 0, 50, 50);
}
```

运行程序，效果如下图所示。

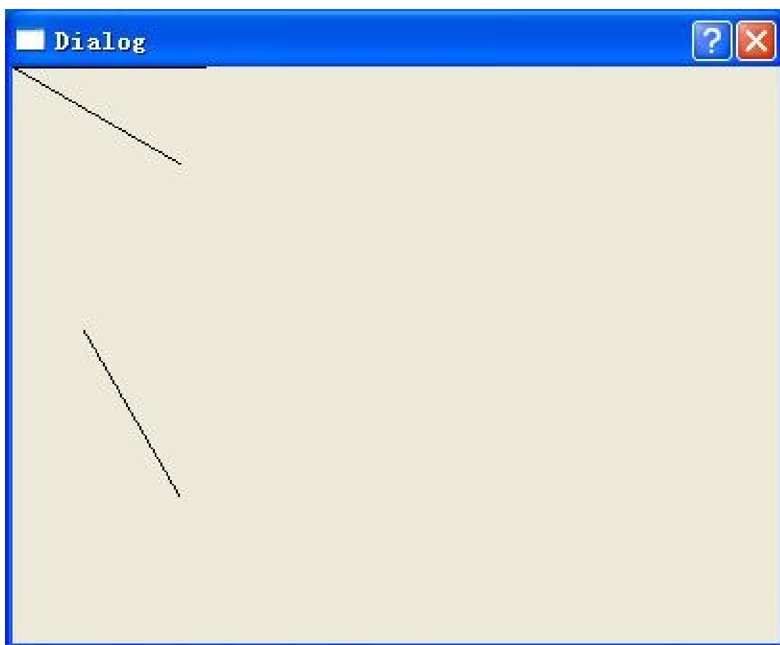


`shear()` 有两个参数，第一个是对横向进行扭曲，第二个是对纵向进行扭曲，而取值就是扭曲的程度。比如程序中对纵向扭曲值为1，那么就是红色正方形左边的边下移一个单位，右边的边下移两个单位，值为1就表明右边的边比左边的边多下移一个单位。大家可以更改取值，测试效果。

4．旋转变换。将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 旋转
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);
    painter.rotate(30); //以原点为中心，顺时针旋转30度
    painter.drawLine(0, 0, 100, 0);
    painter.translate(100, 100);
    painter.rotate(30);
    painter.drawLine(0, 0, 100, 0);
}
```

运行程序，效果如下图所示。

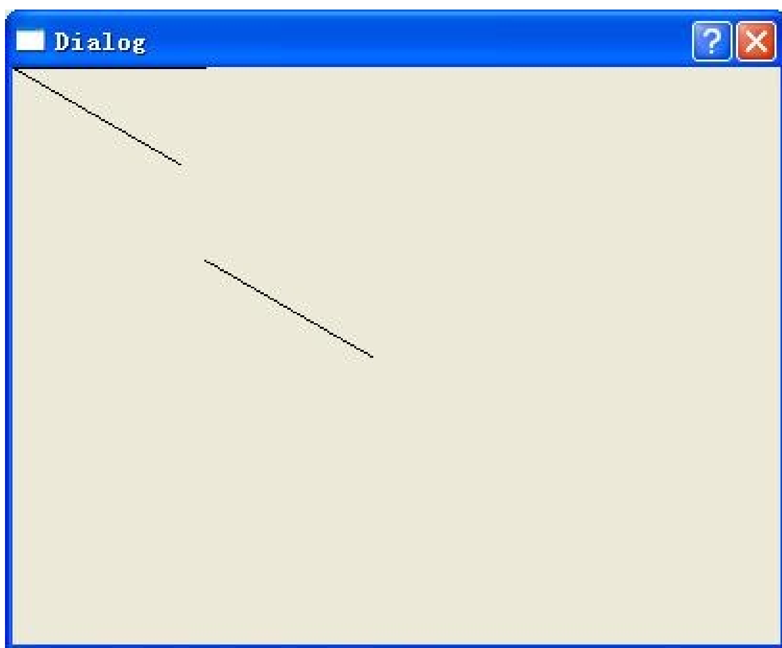


这里先绘制了一条水平的直线，然后将坐标系统旋转了30度，又绘制了一条直线。可以看到，默认是以原点 `(0, 0)` 为中心旋转的。如果想改变旋转中心，可以使用 `translate()` 函数，比如这里将中心移动到了 `(100, 100)` 点，然后旋转了30度，又绘制了一条直线。我们的本意是想在新的原点从水平方向旋转30度进行绘制，可是实际效果却超过了30度。这是由于前面已经使用 `rotate()` 函数旋转过坐标系统了，后面的旋转会在前面的基础上进行。

下面我们再次更改 `paintEvent()` 函数：

```
void Dialog::paintEvent(QPaintEvent *)
{    // 旋转
      QPainter painter(this);
      painter.drawLine(0, 0, 100, 0);
      painter.rotate(30); //以原点为中心，顺时针旋转30度
      painter.drawLine(0, 0, 100, 0);
      painter.rotate(-30); // 反向旋转
      painter.translate(100, 100);
      painter.rotate(30);
      painter.drawLine(0, 0, 100, 0);
}
```

运行程序，效果如下图所示。



这次我们在移动原点以前先将坐标系统反向旋转，可以看到，第二次旋转也是从水平方向开始的。

其实，前面讲到的这几个变换函数都是如此，他们改变了坐标系统以后，如果不进行逆向操作，坐标系统是无法自动复原的。针对这个问题，下面我们将讲解两个非常实用的函数来实现坐标系统的保存和还原。

三、坐标系统的保存

我们可以先利用 `save()` 函数来保存坐标系现在的状态，然后进行变换操作，操作完之后，再用 `restore()` 函数将以前的坐标系状态恢复，其实就是一个入栈和出栈的操作。下面来看一个具体的例子，更改 `paintEvent()` 函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.save(); //保存坐标系状态
    painter.translate(100,100);
    painter.drawLine(0, 0, 50, 50);
    painter.restore(); //恢复以前的坐标系状态
    painter.drawLine(0, 0, 50, 50);
}
```

运行程序，效果如下图所示。利用好这两个函数，可以实现坐标系快速切换，绘制出不同的图形。



第二部分 坐标系统深入研究

在第一部分，我们主要学习了常用的一些坐标变换，虽然在编程中，这些变换已经可以满足大部分的应用需求。不过，大家是否也感觉到现在对坐标的变换依然很模糊，没有一个透彻的认识。下面咱们就一点一点来研究一下坐标系统的变换。

一、获得坐标信息

前面图形的变换都是我们眼睛看到的，为了更具有说服力，下面将获取具体的坐标数据，通过参考数据来进一步了解坐标变换。

1．首先在 `dialog.h` 文件中添加头文件包含：

```
#include <QMouseEvent>
```

然后添加一个 `protected` 鼠标事件处理函数声明：

```
void mousePressEvent(QMouseEvent *);
```

2．到 `dialog.cpp` 文件中，先添加头文件包含：

```
#include <QDebug>
```

然后添加函数定义：

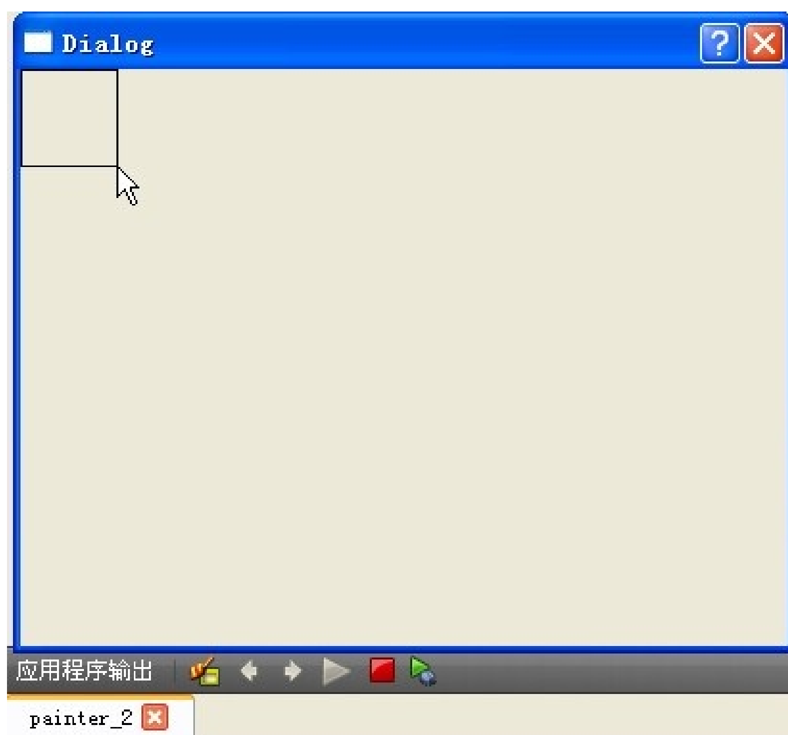
```
void Dialog::mousePressEvent(QMouseEvent *event)
{
         qDebug() << event->pos();
}
```

这里应用了 `qDebug()` 函数，该函数可以在程序运行时将程序中的一些信息输出到控制面板，在QtCreator中会将信息输出到其下面的“应用程序输出”窗口。这个函数很有用，在进行简单的程序调试时，都可以利用该函数进行。我们这里利用它将鼠标指针的坐标值输出出来。

3．将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawRect(0, 0, 50, 50);
}
```

现在运行程序，然后将鼠标在绘制的正方形右下角顶点处点击，在QtCreator的应用程序输出窗口就会输出相应点的坐标信息。如下图所示。大家也可以点击一下其他的地方，查看输出信息。



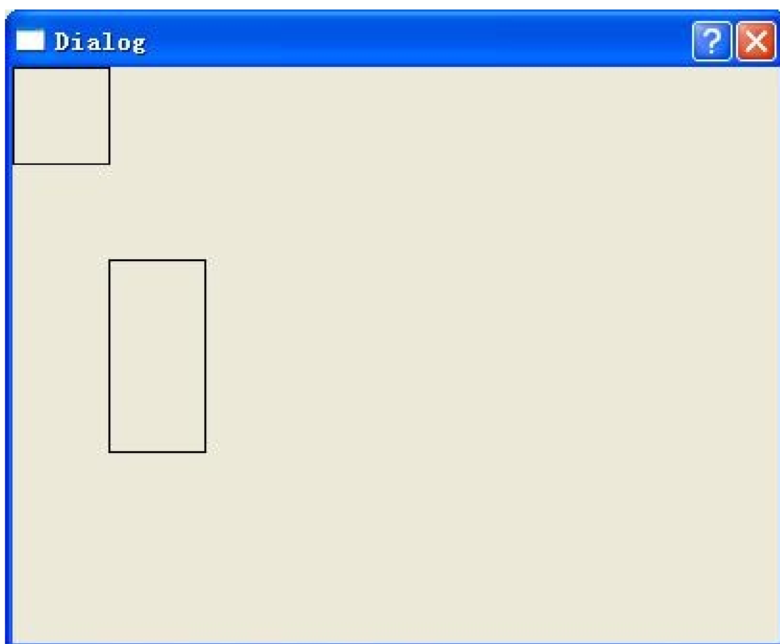

```
E:\painter_2-build-desktop-Debug\debug\painter_2.ex
QPoint(50,50)
```

二、研究变换后的坐标系统

1．首先研究放大后的坐标系统，将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 放大
    QPainter painter(this);
    painter.drawRect(0, 0, 50, 50);
    painter.scale(1, 2);
    painter.drawRect(50, 50, 50, 50);
}
```

这里，我们将纵坐标放大了两倍，而横坐标没有改变。运行程序，效果如下图所示。

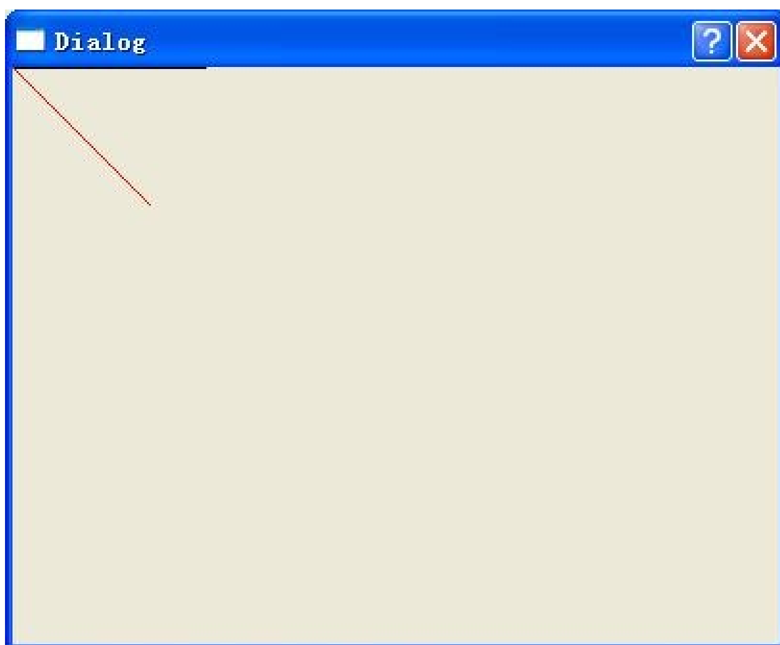


大家可以查看一下第二个矩形的各个顶点的坐标，左上角是 `（50, 100）` 也就是说纵坐标扩大了两倍，查看其它点，会发现左右两条边长都变成了100。

2．研究旋转后的坐标系统。修改 `paintEvent()` 函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);
    painter.rotate(45);
    painter.setPen(Qt::red);
    painter.drawLine(0, 0, 100, 0);
}
```

这里我们先绘制了一条水平的直线，然后将坐标系统旋转45度，再次绘制了一条相同的红色直线。运行程序，效果如下图所示。



大家可以查看一下各处的坐标，虽然旋转后直线位置发生了变化，但是坐标其实是没有变化的。 我们也可以利用这种方法来测试一下应用其他变换函数后坐标的变化，这里就不再敖述。

三、研究绘图设备的坐标系统

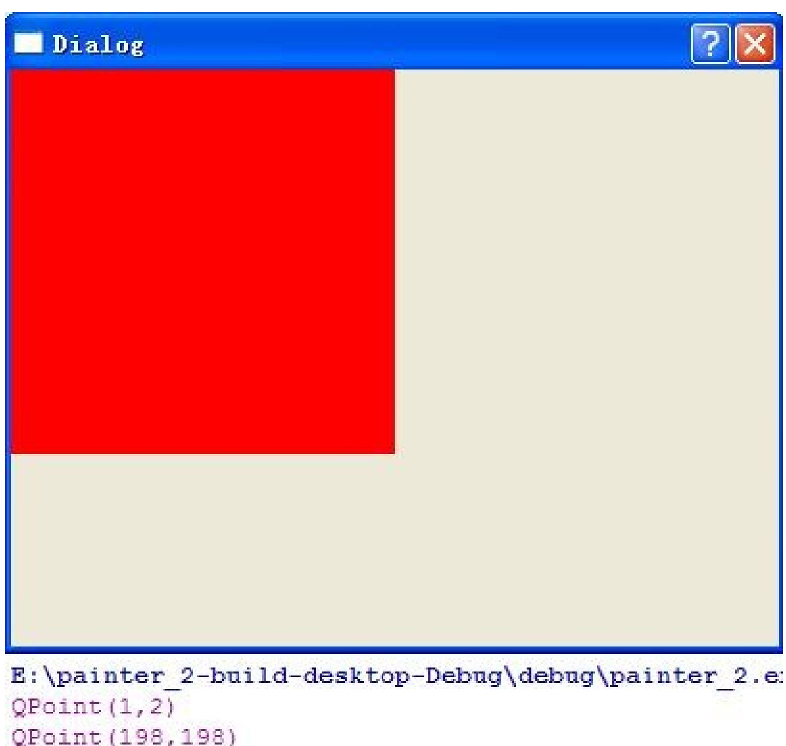
除了可以在 `QWidget` 等窗口部件上进行绘制以外，还可以在 `QPixmap` 、 `QImage` 等上面进行绘制，这些均称为绘图设备。下面我们就以 `QPixmap` 为例，来研究一下它的坐标系统。

1．首先更改 `paintEvent()` 函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);
    pix.fill(Qt::red);    //背景填充为红色
    painter.drawPixmap(0, 0, pix);
}
```

在前面我们已经讲过， `QPixmap` 可以用来显示图片。其实 `QPixmap` 本身就是一个绘图设备，可以在它上面直接绘图。这里先生成了一个宽和高都是200像素的 `QPixmap` 类对象（注意，必须在构建时指定其大小），然后为其填充了红色，最后在窗口的原点进行了绘制。为了表述方面，下面将 `QPixmap` 对象称为画布，这里就是先绘制了一个红色画布。

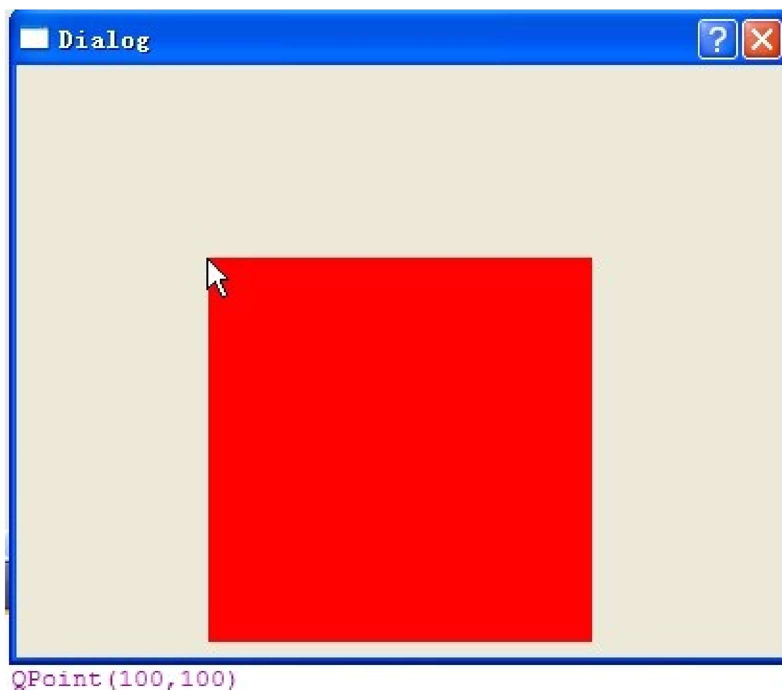
我们运行程序，并在红色画布的左上角和右下角分别点击，查看输出的坐标。如下图所示。因为点击位置的误差，所以两个点可能不是顶点。



2．下面我们接着更改 `paintEvent()` 的代码：

```
void Dialog::paintEvent(QPaintEvent *)
{
        QPainter painter(this);
        QPixmap pix(200, 200);
        pix.fill(Qt::red);    //背景填充为红色
        painter.drawPixmap(100, 100, pix);
}
```

这次我们在 `(100, 100)` 点重新绘制了画布，现在运行程序，发现画布左上角坐标确实为 `(100,100)`，这个就是我们窗口中的坐标，是没有什么疑问的。效果如下图所示。

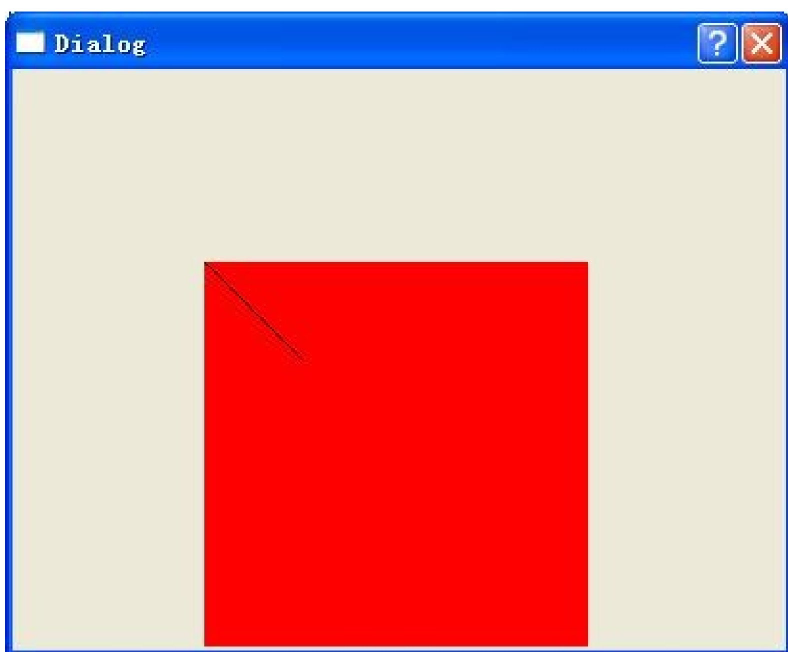


窗口和画布都是绘图设备，那么画布本身有没有自己的坐标系统呢？我们接着研究！

3．我们继续更改 `paintEvent()` 函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);
    pix.fill(Qt::red);
    //新建QPainter类对象，在pix上进行绘图
    QPainter pp(&pix);
    //在pix上的（0，0）点和（50，50）点之间绘制直线
    pp.drawLine(0, 0, 50, 50);
    painter.drawPixmap(100, 100, pix);
}
```

这里我们为画布 `pix` 创建了一个 `QPainter` 对象 `pp`，注意这个 `pp` 只能在画布上绘画，然后我们在画布上绘制了一条从原点 `(0, 0)` 开始的直线。运行程序，效果如下图所示。



可以看到，直线是从画布的左上角开始绘制的，也就是说，画布也有自己的坐标系统，坐标原点在画布的左上角。

下面补充说明一下：`QPainter painter(this)`，`this`就表明了是在窗口上进行绘图，所以利用`painter`进行的绘图都是在窗口部件上的，`painter`进行的坐标变换，是变化的窗口的坐标系；而利用`pp`进行的绘图都是在画布上进行的，如果它进行坐标变化，就是变化的画布的坐标系。

而通过坐标数值，我们可以得出下面两条结论：

第一，`QWidget`和`QPixmap`各有一套坐标系统，它们互不影响。可以看到，无论画布在窗口的什么位置，它的坐标原点依然在左上角，为`（0,0）`点，没有变。

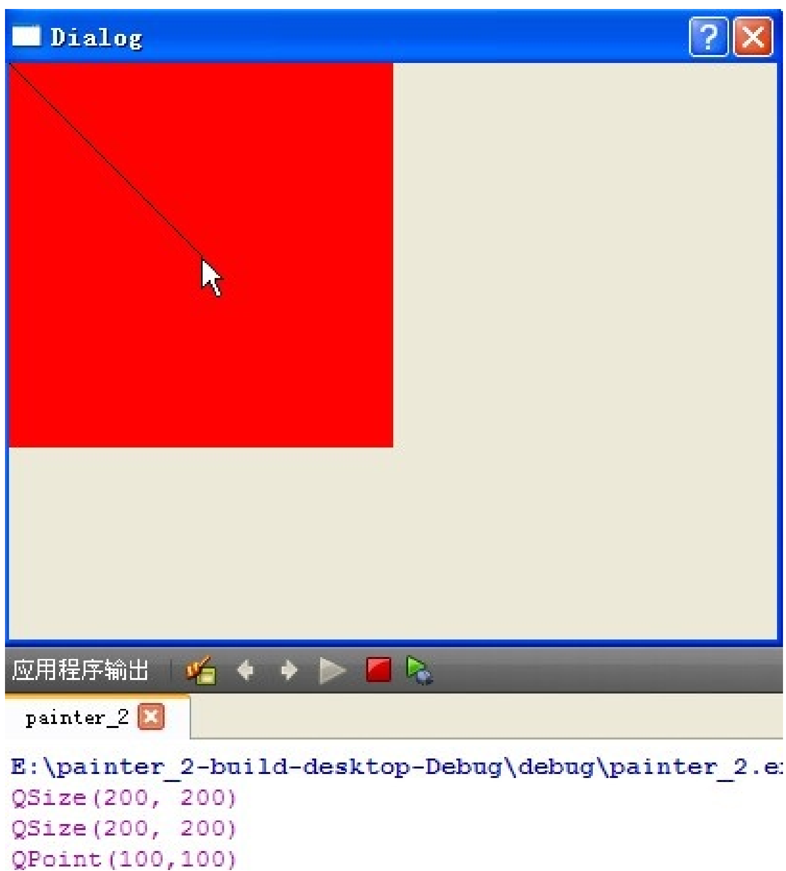
第二，我们所得到的鼠标指针的坐标值是窗口坐标系统的，不是画布的坐标。

4．下面这个例子将对比分析扩大窗口坐标或画布坐标的异同。

首先将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    //放大前输出画布的大小
    qDebug() << pix.size();
    pix.fill(Qt::red);
    QPainter pp(&pix);
    //画布的坐标扩大2倍
    pp.scale(2, 2);
    //在画布上的（0，0）点和（50，50）点之间绘制直线
    pp.drawLine(0, 0, 50, 50);
    //放大后输出画布的大小
    qDebug() << pix.size();
    painter.drawPixmap(0, 0, pix);
}
```

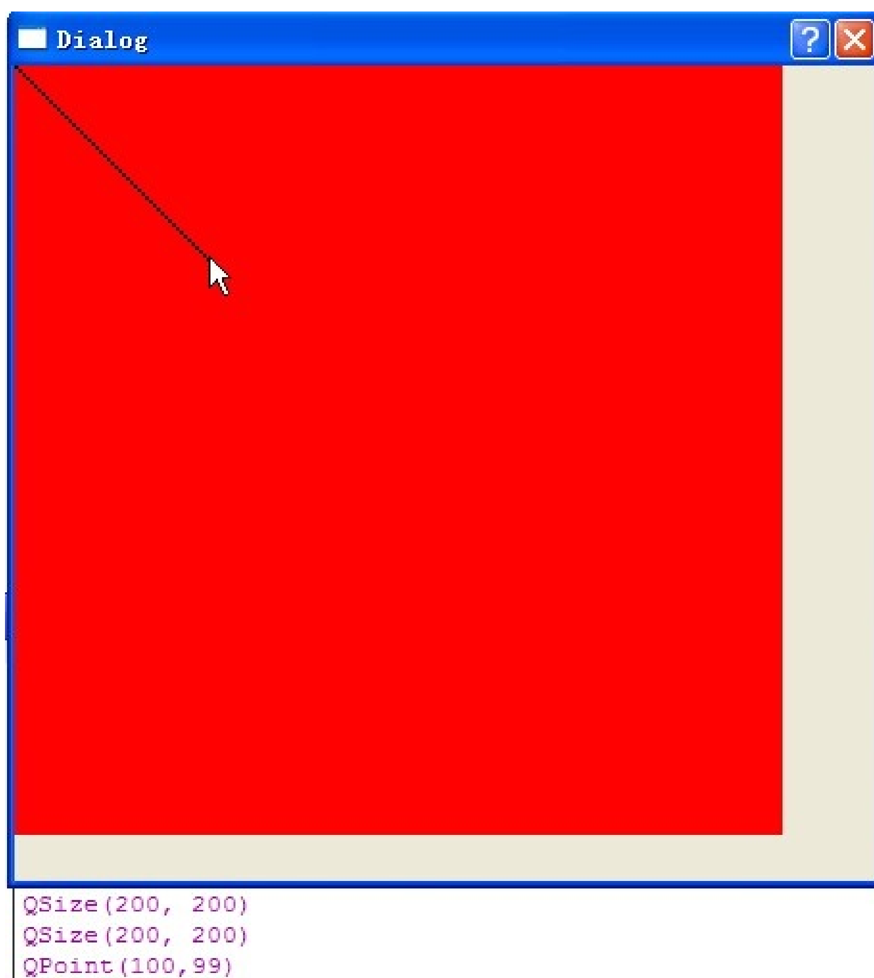
这里我们将画布坐标系统放大了两倍，然后从原点开始绘制了一条直线，并分别输出了画布放大前后的大小。运行程序，效果如下图所示。



下面再次更改 `paintEvent()` 函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    qDebug() << pix.size();
    //窗口坐标扩大2倍
    painter.scale(2,2);
    pix.fill(Qt::red);
    QPainter pp(&pix);
    pp.drawLine(0, 0, 50, 50);
    qDebug() << pix.size();
    painter.drawPixmap(0, 0, pix);
}
```

这里与前面唯一的不同是：这里放大了窗口的坐标系统，而前面放大的是画布的坐标系统。运行程序，效果如下图所示。



可以看到，整个画布的可见面积变大了。直线虽然长度依然是100，但是这次的效果跟前面明显不同，因为是窗口坐标变大，所以在上面绘出的线条有了明显的颗粒感。

上面两个程序虽然最终输出的数据是一样的，但实际效果还是有很大不同的。大家可以根据需要进行选择性应用。

结语

在这一节中我们讲述了坐标相关的多个知识点，经过本节的学习，大家应该已经对Qt的2D绘图有了一个浅显的认识，下一节我们将做一个比较实用的涂鸦板例子。

Qt的坐标系统是很有必要好好研究的，它对深入学习应用Qt绘图很有帮助。如果大家想更系统的学习Qt坐标系统，可以参考《Qt Creator快速入门》的第10章相关内容。

涉及到的源码：

- painter_2_1.zip
- painter_2_2.zip
- painter_2_3.zip

第17篇 2D绘图（七）涂鸦板

导语

通过前面几节的学习，大家应该已经对Qt中2D绘图有了一定的认识，这一节我们将应用前面讲到的内容，编写一个简单的涂鸦板程序，这一节只是实现最基本的鼠标画线功能。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目 录

正文

一、实现涂鸦板

1．新建Qt Gui应用，项目名称为 `pianter_3` ，基类这次还用 `QDialog` ，类名保持 `Dialog` 不变即可。

2．到 `dialog.h` 文件中，先添加头文件包含： `#include <QMouseEvent>`

然后添加几个函数的声明：

```
protected:
    void paintEvent(QPaintEvent *);
    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
    void mouseReleaseEvent(QMouseEvent *);
```

第一个是绘制事件处理函数，后面分别是鼠标按下、移动和释放事件的处理函数。

下面再添加几个 `private` 私有变量声明：

```
QPixmap pix;
QPoint lastPoint;
QPoint endPoint;
```

因为在函数里声明的 `QPixmap` 类对象是临时变量，不能存储以前的值，为了实现保留上次的绘画结果，我们需要将其设为全局变量。后面两个 `QPoint` 变量存储鼠标指针的两个坐标值，我们需要用这两个坐标值完成绘图。

2、到 `dialog.cpp` 文件中，先添加头文件包含： `#include <QPainter>`

然后在构造函数中添加如下初始代码：

```
resize(600, 500);    //窗口大小设置为600*500
pix = QPixmap(200, 200);
pix.fill(Qt::white);
```

下面添加几个函数的定义：

```
void Dialog::paintEvent(QPaintEvent *)
{
      QPainter pp(&pix);    // 根据鼠标指针前后两个位置就行绘制直线
      pp.drawLine(lastPoint, endPoint);    // 让前一个坐标值等于后一个坐标值，这样就能实现画出
连续的线
      lastPoint = endPoint;
      QPainter painter(this);
      painter.drawPixmap(0, 0, pix);
}
```

这里使用了两个点来绘制线条，这两个点在下面的鼠标事件中获得。

```
void Dialog::mousePressEvent(QMouseEvent *event)
{
               if(event->button()==Qt::LeftButton) //鼠标左键按下
               lastPoint = event->pos();
}
```

当鼠标左键按下时获得开始点。

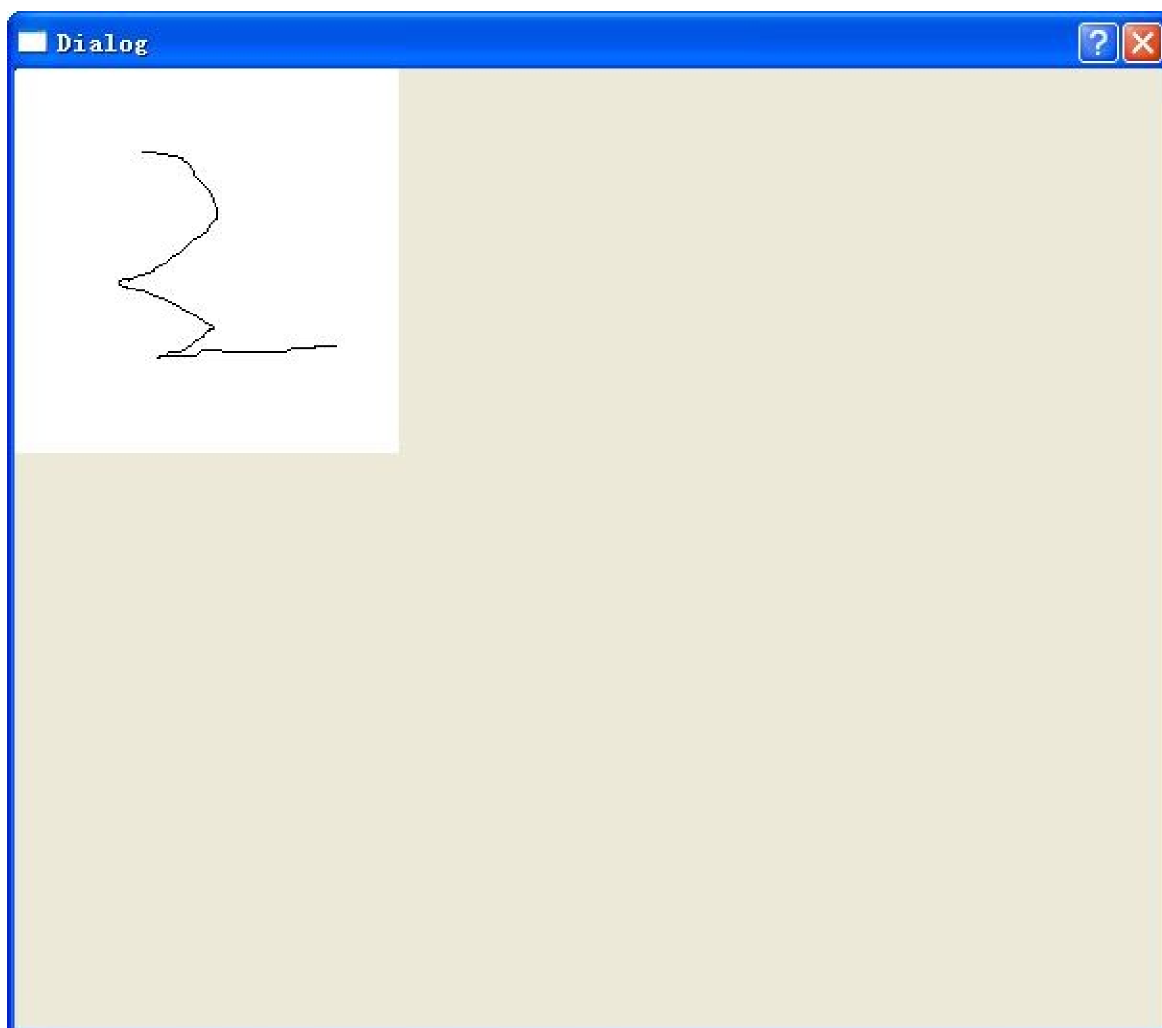
```
void Dialog::mouseMoveEvent(QMouseEvent *event)
{
         if(event->buttons()&Qt::LeftButton) //鼠标左键按下的同时移动鼠标
        {
                  endPoint = event->pos();
                  update(); //进行绘制
         }
}
```

当鼠标移动时获得结束点，并更新绘制。调用 `update()` 函数会执行 `paintEvent()` 函数进行重新绘制。

```
void Dialog::mouseReleaseEvent(QMouseEvent *event)
{
            if(event->button() == Qt::LeftButton) //鼠标左键释放
             {
                     endPoint = event->pos();
                     update();
             }
}
```

当鼠标按键释放时也进行重绘。

现在运行程序，使用鼠标在白色画布上进行绘制，发现已经实现了简单的涂鸦板功能，效果如下图所示。



二、实现放大功能

前面已经实现了简单的绘制功能，下面我们将实现放大功能，将画布放大后继续进行涂鸦。这里将使用两种方法来实现，也是对上一节坐标系统后面的问题的更进一步的应用实践。

1．添加放大按钮。到 `dialog.h` 文件中，先添加头文件：

```
#include <QPushButton>
```

然后添加下面 `private` 私有变量声明：

```
qreal scale;
QPushButton *button;
```

最后再添加一个私有槽声明：

```
private slots:
    void zoomIn();
```

2．到 `dialog.cpp` 文件中，先在构造函数中添加如下代码：

```
//设置初始放大倍数为1，即不放大
scale =1;
//新建按钮对象
button = new QPushButton(this);
//设置按钮显示文本
button->setText(tr("zoomIn"));
//设置按钮放置位置
button->move(500, 450);
//对按钮的单击事件和其槽函数进行关联
connect(button, SIGNAL(clicked()), this, SLOT(zoomIn()));
```

这里使用代码创建了一个按钮对象，并将其单击信号关联到了放大槽上，也就是说按下这个按钮，就会执行 `zoomIn()` 槽。

3．下面添加 `zoomIn()` 的定义：

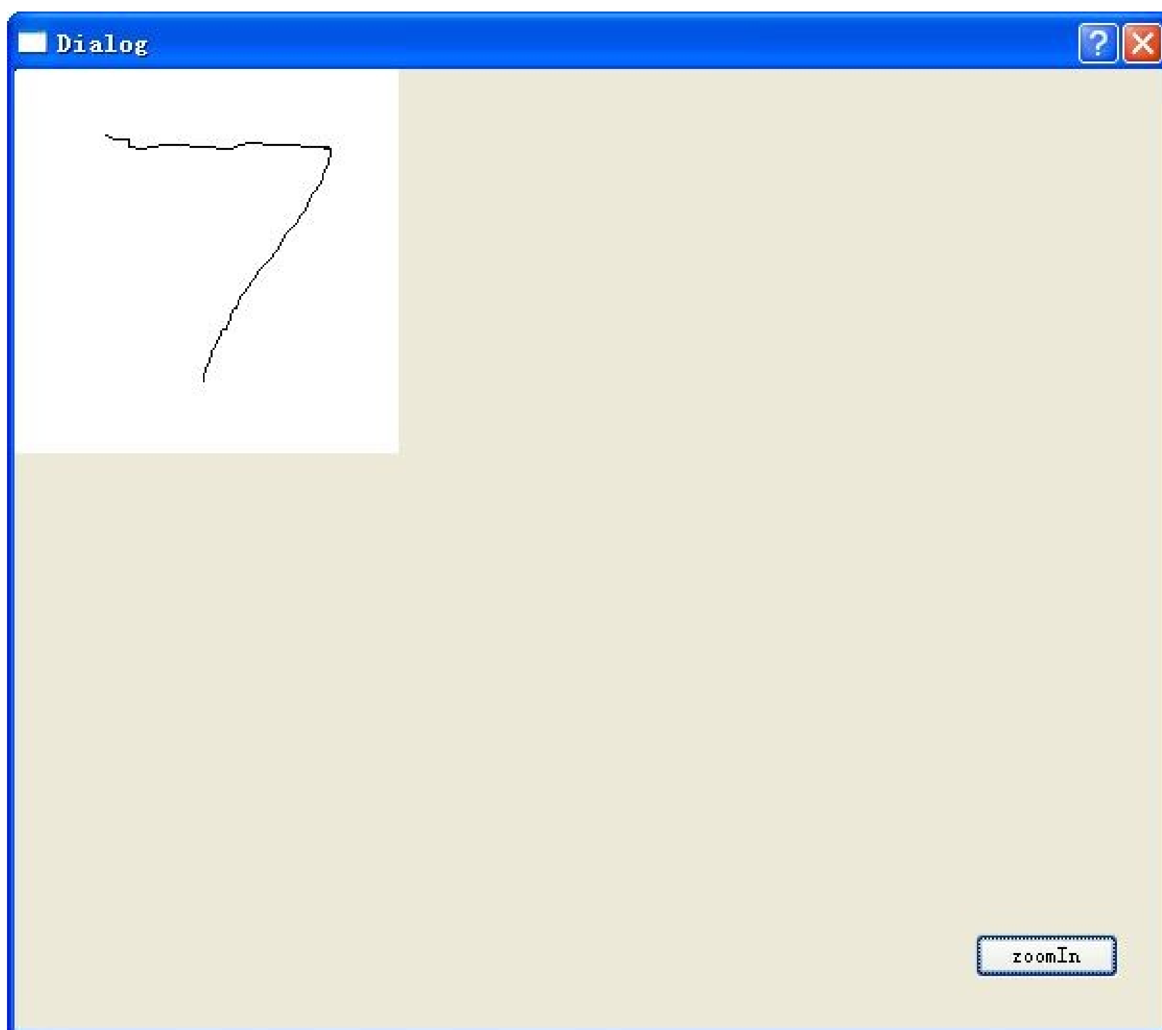
```
void Dialog::zoomIn()
{
    scale *=2;
    update();
}
```

这里我们让每按下这个按钮，放大值都扩大两倍。后面调用 `update()` 函数来更新显示。

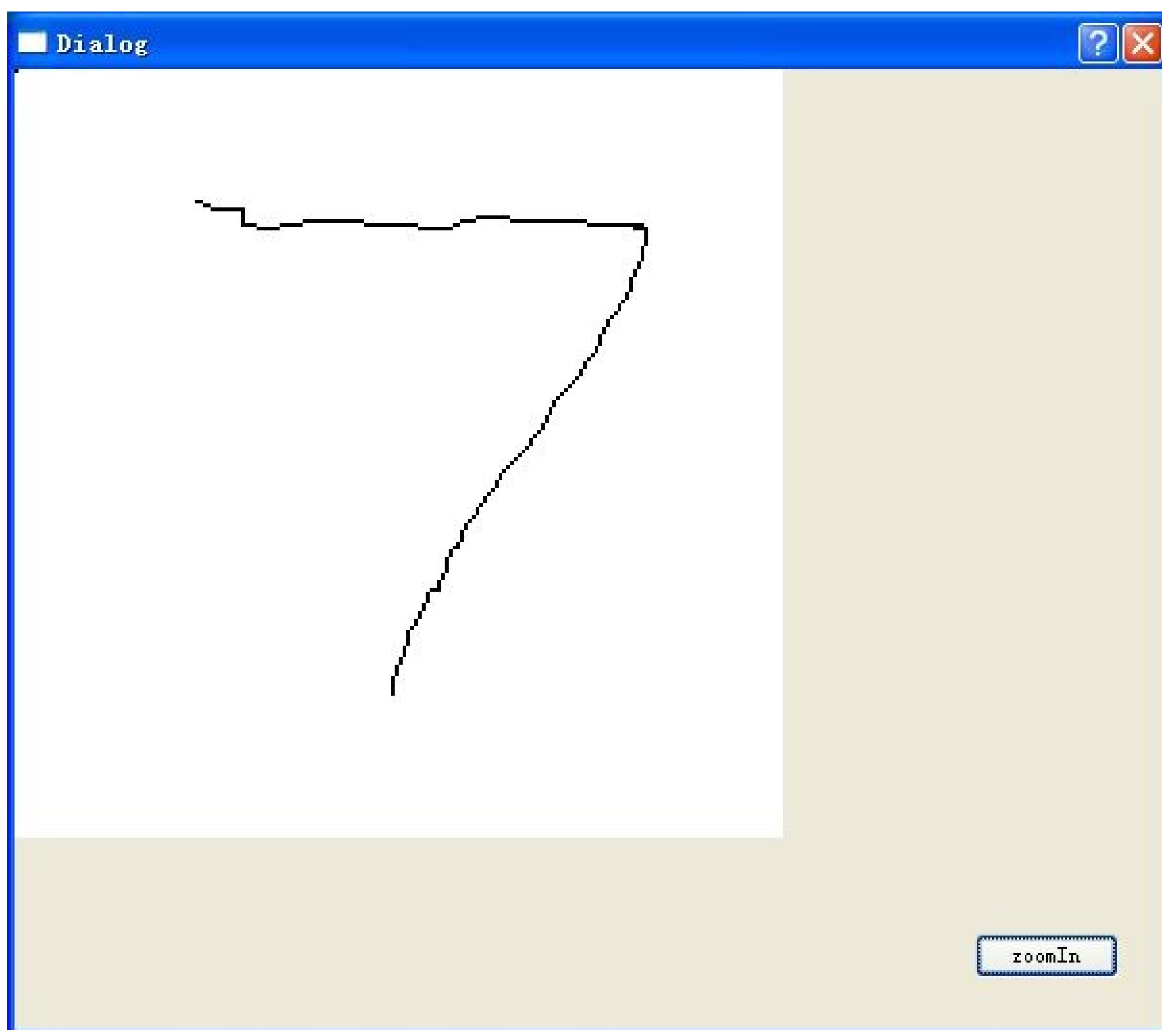
4．通过上一节的学习，我们应该已经知道想让画布的内容放大有两个办法，一个是直接放大画布的坐标系统，一个是放大窗口的坐标系统。下面我们先来放大窗口的坐标系统。更改 `paintEvent()` 函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter pp(&pix);
    pp.drawLine(lastPoint, endPoint);
    lastPoint = endPoint;
    QPainter painter(this);
    //进行放大操作
    painter.scale(scale, scale);
    painter.drawPixmap(0, 0, pix);
}
```

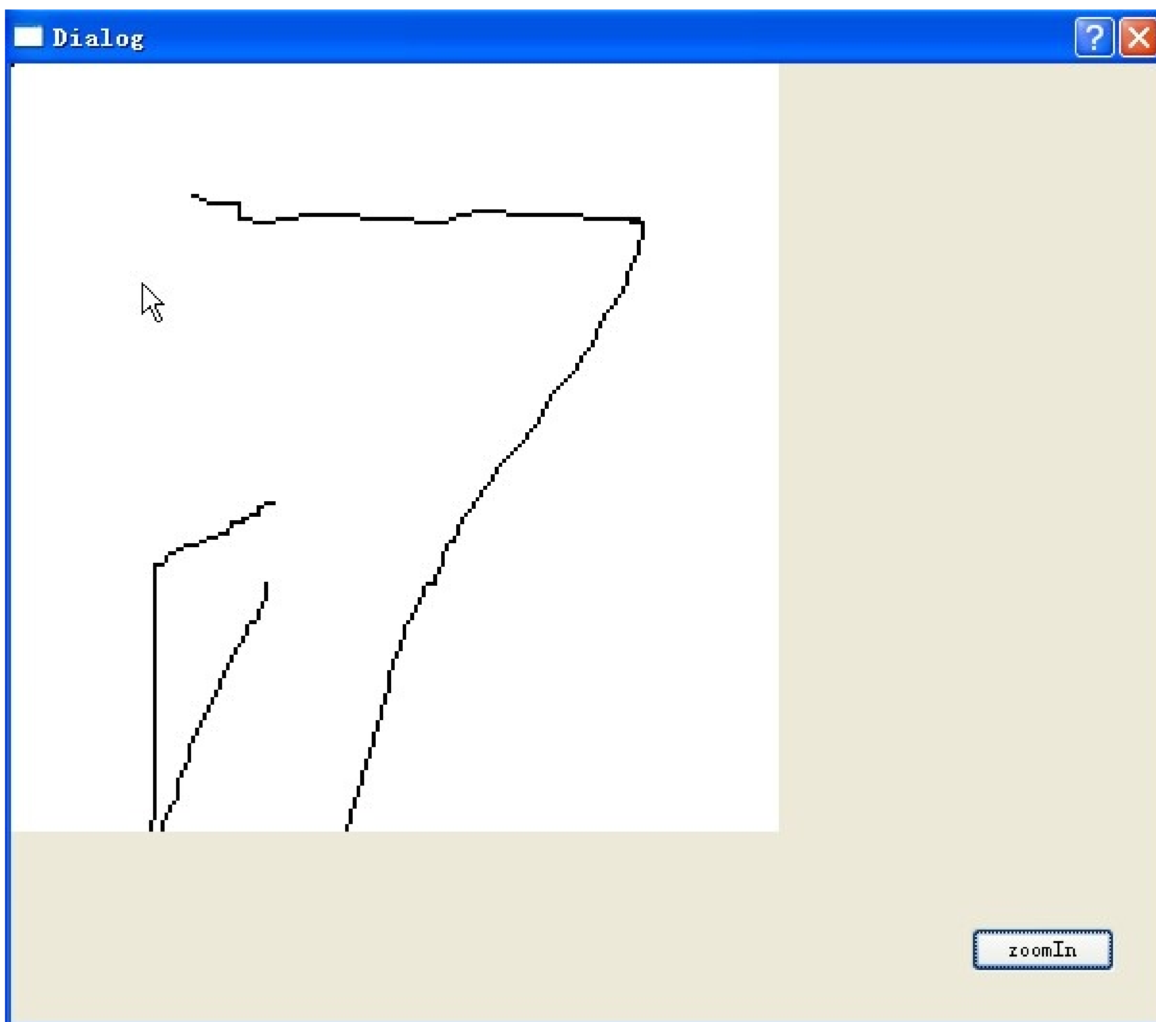
现在运行程序，先在白色画布上任意绘制一个图形，效果如下图所示。



然后按下 `zoomIn` 按钮，效果如下图所示。



现在再用鼠标进行绘制，发现图形已经不能和鼠标轨迹重合了，效果如下图所示。

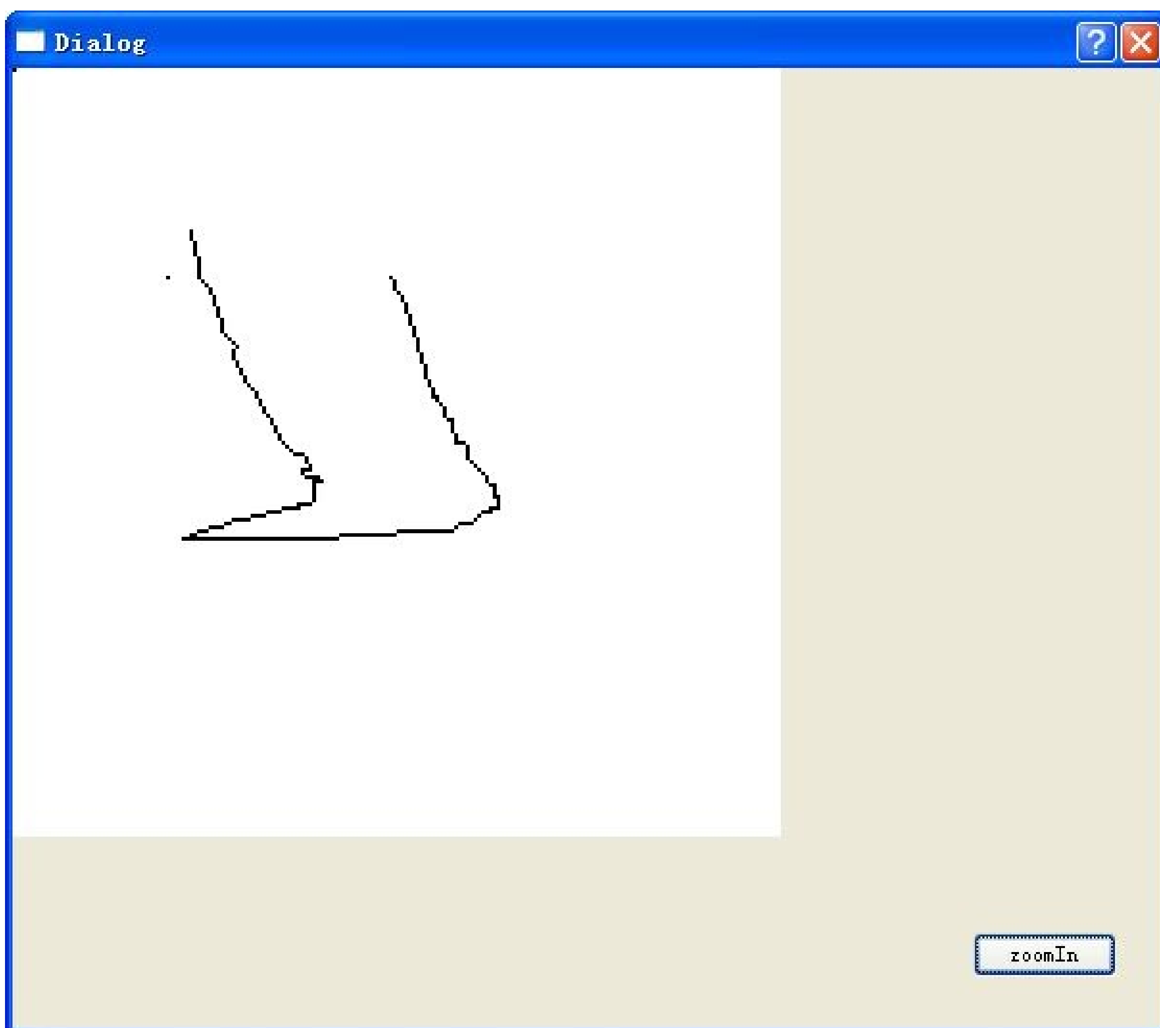


有了前面一节的知识，就不难理解出现这个问题的原因了。窗口的坐标扩大了，但是画布的坐标并没有扩大，而我们画图用的坐标值是鼠标指针的，鼠标指针又是获取的窗口的坐标值。现在窗口和画布的同一点的坐标并不相等，所以就出现了这样的问题。

其实解决办法很简单，窗口放大了多少倍，就将获得的鼠标指针的坐标值缩小多少倍就行了。我们将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter pp(&pix);
    pp.drawLine(lastPoint/scale, endPoint/scale);
    lastPoint = endPoint;
    QPainter painter(this);
    painter.scale(scale, scale);
    painter.drawPixmap(0, 0, pix);
}
```

运行程序，效果如下图所示。可以看到，已经能够在放大以后继续绘图了。

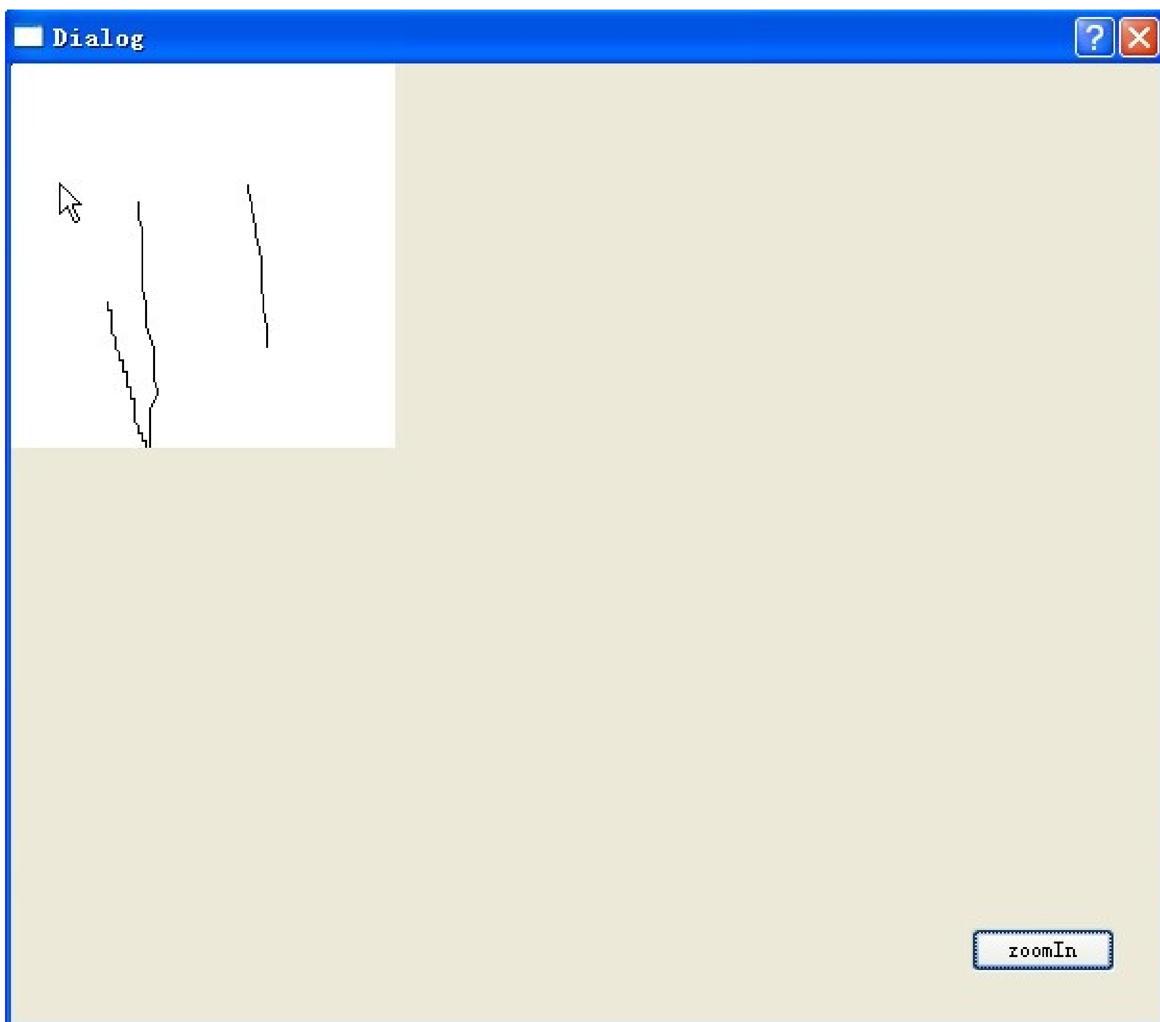


这种用改变窗口坐标大小来改变画布面积的方法，实际上是有损图片质量的。就像将一张位图放大一样，越放大越不清晰。原因就是，它的像素的个数没有变，如果将可视面积放大，那么单位面积里的像素个数就变少了，所以画质就差了。

5．方法二。扩大画布坐标系统。先将 `paintEvent()` 更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter pp(&pix);
    pp.scale(scale, scale);
    pp.drawLine(lastPoint,endPoint);
    lastPoint = endPoint;
    QPainter painter(this);
    painter.drawPixmap(0, 0, pix);
}
```

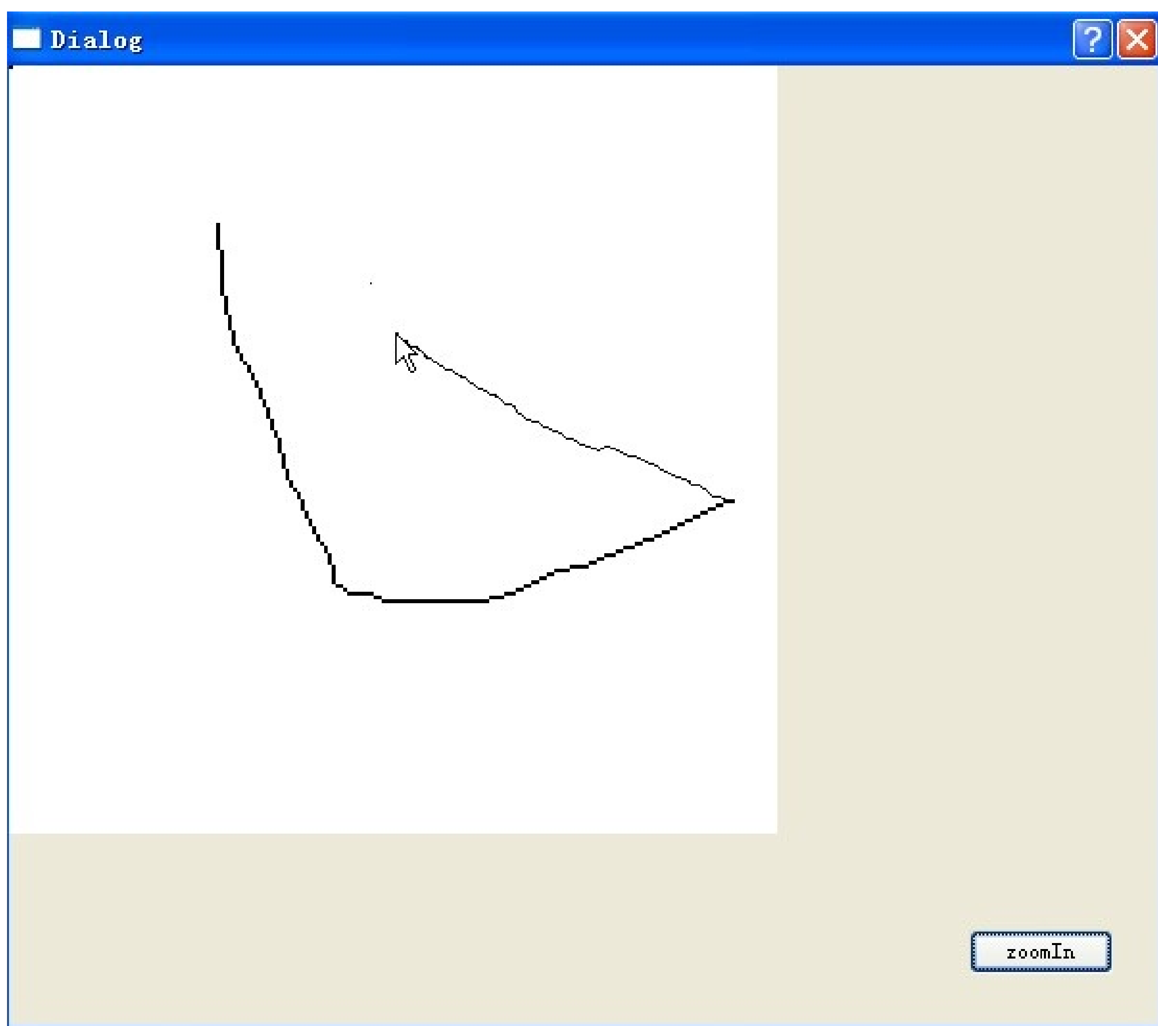
这时运行程序，先进行绘制，然后点击 `zoomIn` 按钮，发现以前的内容并没有放大，而当我们再次绘画时，发现鼠标指针和绘制的线条又不重合了。效果如下图所示。



这并不是我们想要的结果，为了实现按下放大按钮，画布和图形都进行放大，我们可以使用缓冲画布（就是一个辅助画布）来实现。将 `paintEvent()` 函数内容更改如下。

```
void Dialog::paintEvent(QPaintEvent *)
{
    if(scale!=1) //如果进行放大操作
    {
       //临时画布，大小变化了scale倍
       QPixmap copyPix(pix.size()*scale);
       QPainter pter(&copyPix);
       pter.scale(scale, scale);
       //将以前画布上的内容复制到现在的画布上
       pter.drawPixmap(0, 0, pix);
       //将放大后的内容再复制回原来的画布上
       pix = copyPix;
       //让scale重新置1
       scale =1;
    }
    QPainter pp(&pix);
    pp.scale(scale,scale);
    pp.drawLine(lastPoint/scale,endPoint/scale);
    lastPoint = endPoint;
    QPainter painter(this);
    painter.drawPixmap(0,0,pix);
}
```

现在运行程序，效果如下图所示。



结语

本节讲到的涂鸦板，只是为了实践前面的知识，起到抛砖引玉的作用。大家可以根据自己的理解继续探究下去。在下一节，我们将讲解怎样在涂鸦板上绘制出矩形、椭圆等图形。

本程序中存在一些问题，如果大家想进一步学习研究，可以参考下载页面的涂鸦板开源软件。

涉及到的源码

第18篇 2D绘图（八）双缓冲绘图

导语

在前面一节中，讲述了如何实现简单的涂鸦板，这一次我们将实现在涂鸦板上绘制图形，这里以矩形为例进行讲解。在后面还会提出双缓冲绘图的概念。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

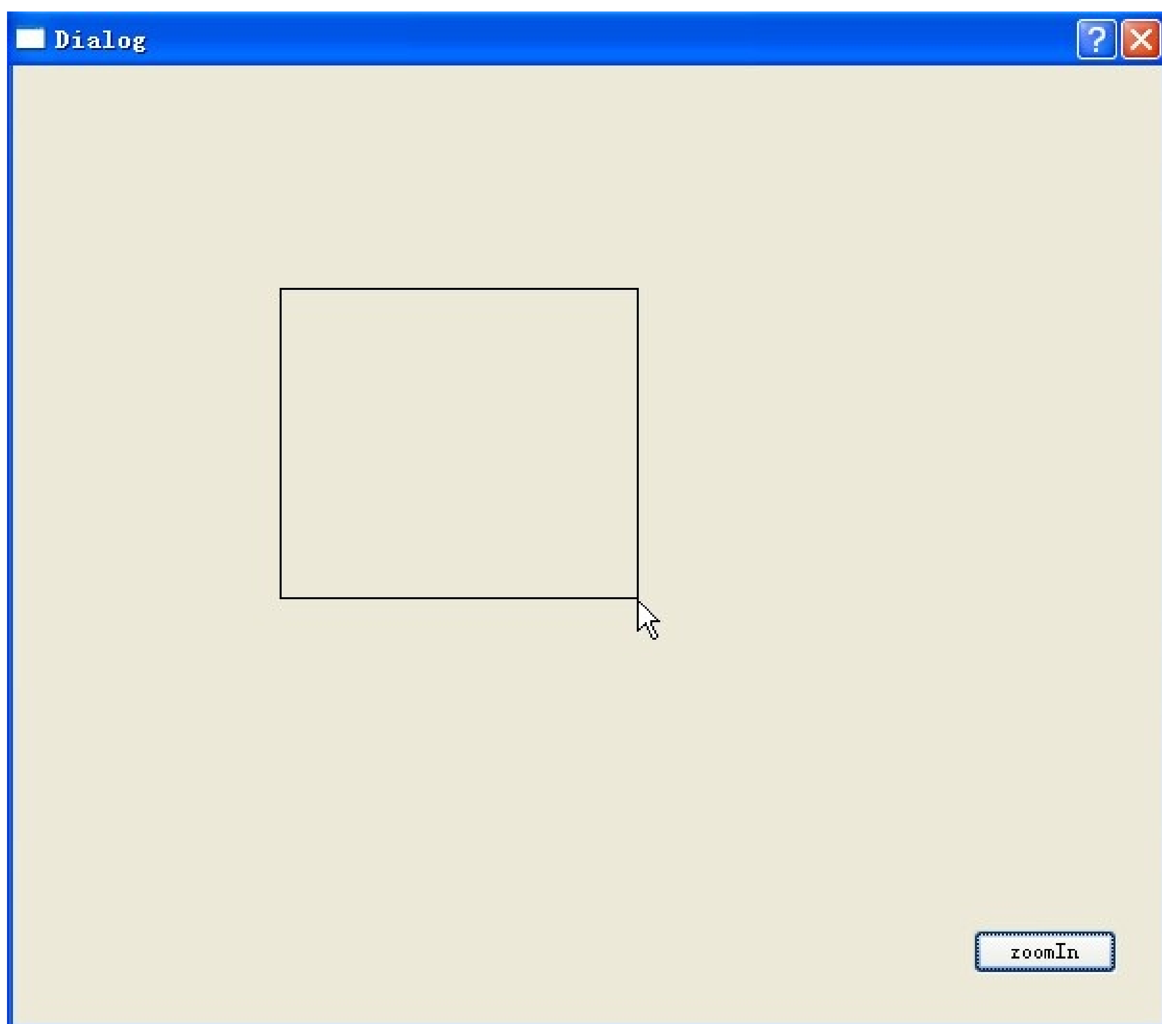
正文

一、绘制矩形

1．我们仍然在前面程序的基础上进行修改，先更改 `painEvent()` 函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    int x,y,w,h;
    x = lastPoint.x();
    y = lastPoint.y();
    w = endPoint.x() - x;
    h = endPoint.y() - y;
    painter.drawRect(x, y, w, h);
}
```

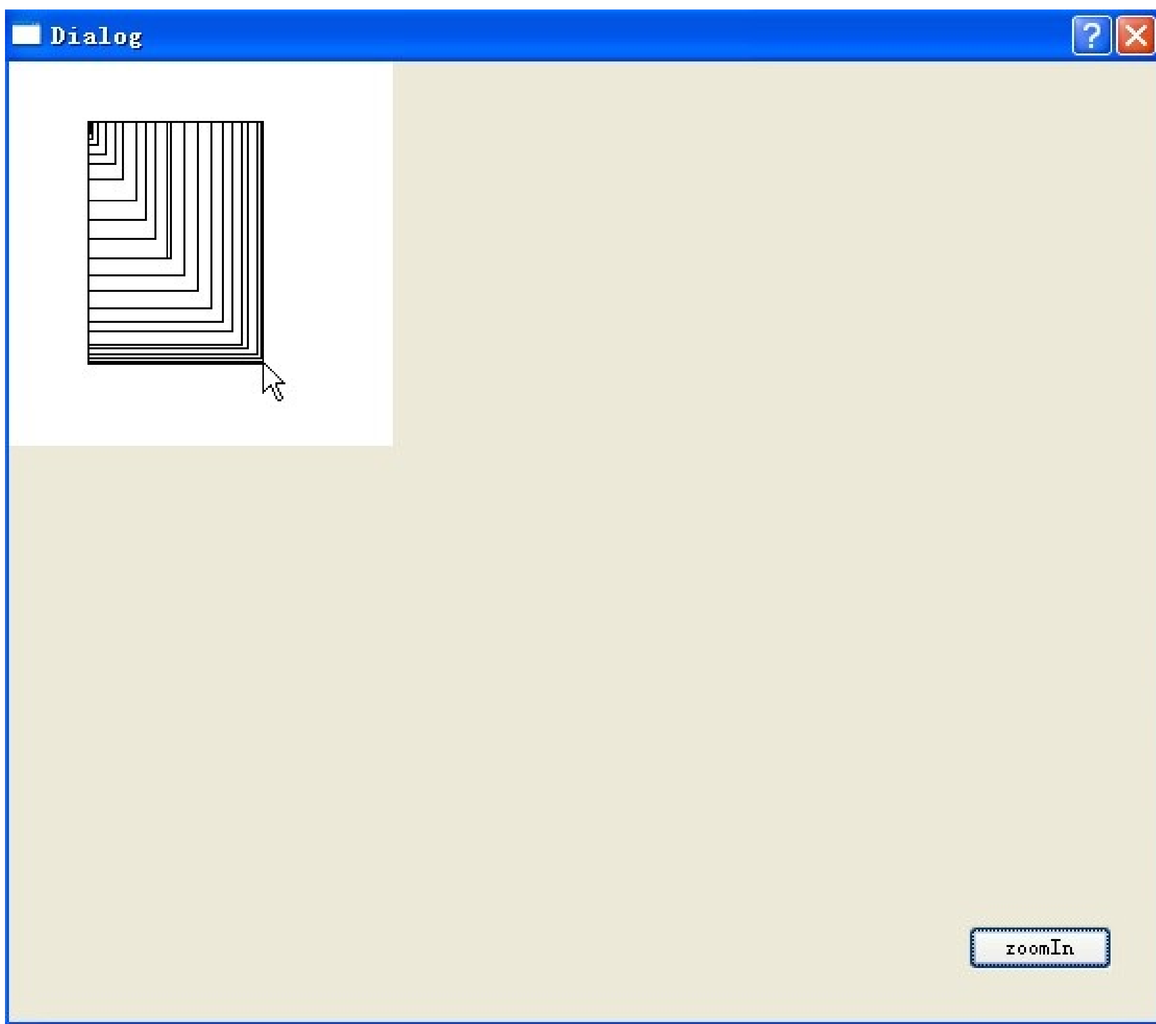
这里就是通过lastPoint和endPoint两个点来确定要绘制的矩形的起点、宽和高的。运行程序，用鼠标拖出一个矩形，效果如下图所示。



2．上面已经可以拖出一个矩形了，但是这样直接在窗口上绘图，以前画的矩形是不能保存下来的。所以我们下面加入画布，在画布上进行绘图。将 `paintEvent()` 函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    int x,y,w,h;
    x = lastPoint.x();
    y = lastPoint.y();
    w = endPoint.x() - x;
    h = endPoint.y() - y;
    QPainter pp(&pix);
    pp.drawRect(x, y, w, h);
    QPainter painter(this);
    painter.drawPixmap(0, 0, pix);
}
```

这里就是将图形先绘制在了画布上，然后将画布绘制到窗口上。我们运行程序，然后使用鼠标拖出一个矩形，发现出现了很多重影，效果如下图所示。



为什么会出现这种现象呢？大家可以尝试分别快速拖动鼠标和慢速拖动鼠标来绘制矩形，结果会发现，拖动速度越快，重影越少。其实，在我们拖动鼠标的过程中，屏幕已经刷新了很多次，也可以理解为 `paintEvent()` 函数执行了多次，每执行一次就会绘制一个矩形。知道了原因，就有方法来避免这个问题发生了。

二、双缓冲绘图

1. 我们再添加一个辅助画布，如果正在绘图，也就是鼠标按键还没有释放的时候，就在这个辅助画布上绘图，只有当鼠标按键释放的时候，才在真正的画布上绘图。

首先在 `dialog.h` 文件中添加两个私有变量：

```
QPixmap tempPix; //辅助画布
bool isDrawing;   //标志是否正在绘图
```

然后到 `dialog.cpp` 的构造函数中对变量进行初始化：

```
isDrawing = false;
```

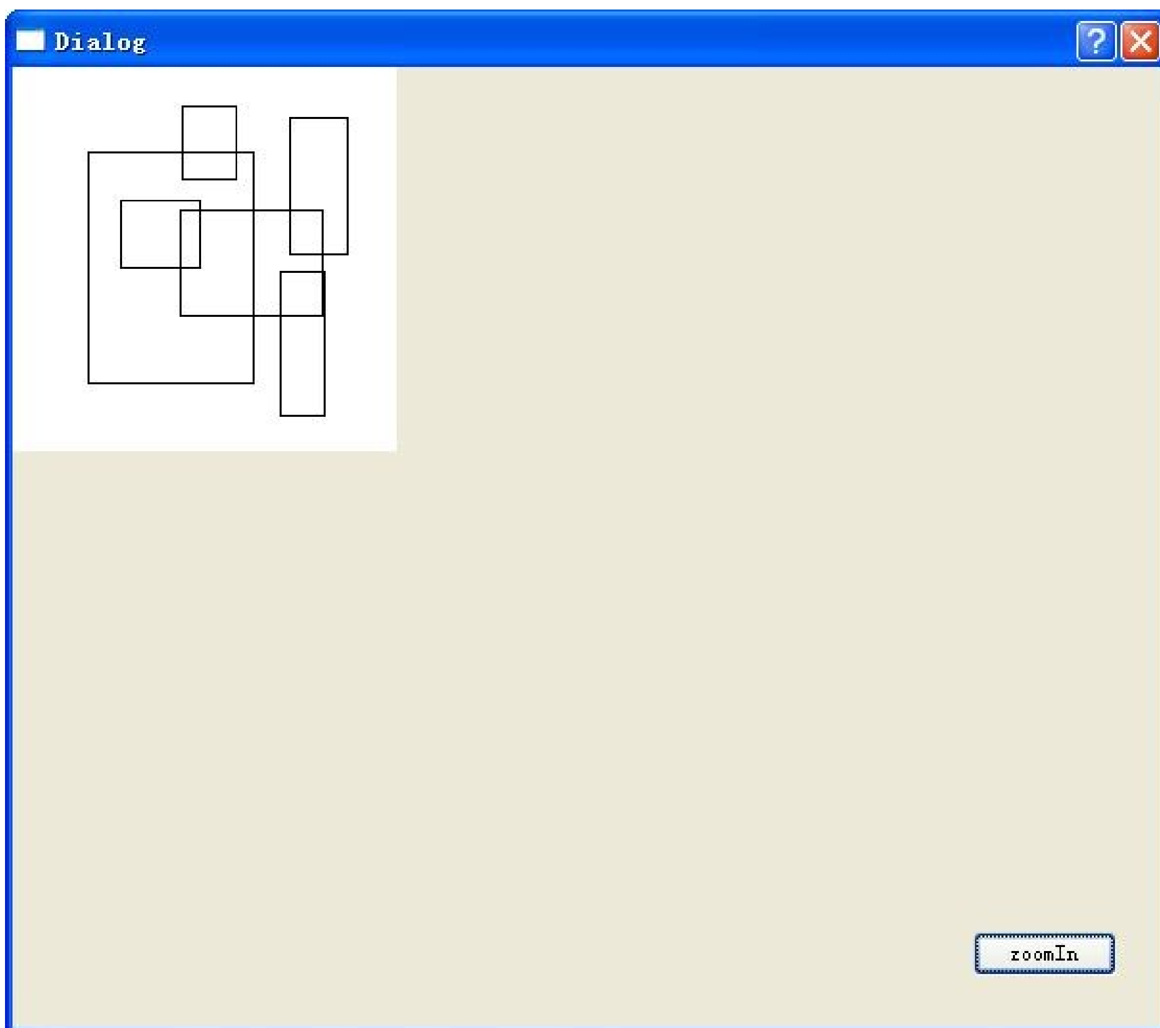
下面再更改 `paintEvent()` 函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    int x,y,w,h;
    x = lastPoint.x();
    y = lastPoint.y();
    w = endPoint.x() - x;
    h = endPoint.y() - y;
    QPainter painter(this);
    if(isDrawing) //如果正在绘图，就在辅助画布上绘制
    {
       //将以前pix中的内容复制到tempPix中，保证以前的内容不消失
       tempPix = pix;
       QPainter pp(&tempPix);
       pp.drawRect(x,y,w,h);
       painter.drawPixmap(0, 0, tempPix);
    } else {
       QPainter pp(&pix);
       pp.drawRect(x,y,w,h);
       painter.drawPixmap(0,0,pix);
    }
}
```

下面还需要更改鼠标按下事件处理函数和鼠标释放事件处理函数的内容：

```
void Dialog::mousePressEvent(QMouseEvent *event)
{
    if(event->button()==Qt::LeftButton) //鼠标左键按下
    {
       lastPoint = event->pos();
       isDrawing = true;    //正在绘图
    }
}
void Dialog::mouseReleaseEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton) //鼠标左键释放
    {
       endPoint = event->pos();
       isDrawing = false;    //结束绘图
       update();
    }
}
```

当鼠标左键按下时我们开始标记正在绘图，当按键释放时我们取消正在绘图的标记。现在运行程序，已经可以实现正常的绘图了。效果如下图所示。



2．双缓冲绘图

根据这个例子所使用的技巧，我们引出所谓的双缓冲绘图的概念。双缓冲（double-buffers）绘图，就是在进行绘制时，先将所有内容都绘制到一个绘图设备（如 `QPixmap` ）上，然后再将整个图像绘制到部件上显示出来。使用双缓冲绘图可以避免显示时的闪烁现象。从Qt 4.0开始， `QWidget` 部件的所有绘制都自动使用了双缓冲，所以一般没有必要在 `paintEvent()` 函数中使用双缓冲代码来避免闪烁。

虽然在一般的绘图中无需手动使用双缓冲绘图，不过要想实现一些绘图效果，还是要借助于双缓冲的概念。比如这个程序里，我们要实现使用鼠标在界面上绘制一个任意大小的矩形。这里需要两张画布，它们都是 `QPixmap` 实例，其中一个 `tempPix` 用来作为临时缓冲区，当鼠标正在拖动矩形进行绘制时，将内容先绘制到 `tempPix` 上，然后将 `tempPix` 绘制到界面上；而另一个 `pix` 作为缓冲区，用来保存已经完成的绘制。当松开鼠标完成矩形的绘制后，则将 `tempPix` 的内容复制到pix上。为了绘制时不显示拖影，而且保证以前绘制的内容不消失，那么在移动鼠标过程中，每绘制一次，都要在绘制这个矩形的原来的图像上进行绘制，所以需要在每次绘制 `tempPix` 之前，先将pix的内容复制到 `tempPix` 上。因为这里有两个 `QPixmap` 对象，也可以说有两个缓冲区，所以称之为双缓冲绘图。

结语

对于Qt基本绘图的内容，我们就讲到这里，如果大家还想更加系统深入的学习这些基础知识，可以参考《Qt Creator快速入门》的第10章。从下一节开始，我们将简单介绍一下Qt中得图形视图框架。

涉及到的源码

第19篇 2D绘图（九）图形视图框架（上）

导语

在前面讲的基本绘图中，我们可以自己绘制各种图形，并且控制它们。但是，如果需要同时绘制很多个相同或不同的图形，并且要控制它们的移动，检测它们的碰撞和叠加；或者我们想让自己绘制的图形可以拖动位置，进行缩放和旋转等操作。实现这些功能，要是还使用以前的方法，那么会十分困难。解决这些问题，可以使用Qt提供的图形视图框架。

图形视图可以对大量定制的2D图形项进行管理和相互作用。视图部件可以让所有图形项可视化，它还提供了缩放和旋转功能。我们在帮助中搜索 `Graphics View` 关键字，内容如下图：

Graphics View provides a surface for managing and interacting with a large number of custom-made 2D graphical items, and a view widget for visualizing the items, with support for zooming and rotation.

The framework includes an event propagation architecture that allows precise double-precision interaction capabilities for the items on the scene. Items can handle key events, mouse press, move, release and double click events, and they can also track mouse movement.

Graphics View uses a BSP (Binary Space Partitioning) tree to provide very fast item discovery, and as a result of this, it can visualize large scenes in real-time, even with millions of items.

Graphics View was introduced in Qt 4.2, replacing its predecessor, QCanvas. If you are porting from QCanvas, see Porting to Graphics View.

Topics:

The Graphics View Architecture

这里一开始对这个框架进行了简单介绍，整个图形视图结构主要包含三部分：场景（ `Scene` ）、视图（ `View` ）和图形项（ `Item` ），它们分别对应 `QGraphicsScene` 、 `QGraphicsView` 、QGraphicsItem三个类。其实图形视图框架是一组类的集合，在帮助中可以看到所有与它相关的类。下面我们就开始结合程序对整个框架进行介绍。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

- （三）拖放
- （四）键盘与鼠标事件
- （五）碰撞检测
- （六）移动
- （七）动画
- （八）右键菜单

正文

一、基本应用

我们新建空的Qt项目（在其他项目中），项目名称为 `graphicsView01` 。然后在这个项目中添加新的C++ 源文件，命名为 `main.cpp` 。

我们将 `main.cpp` 的内容更改如下。

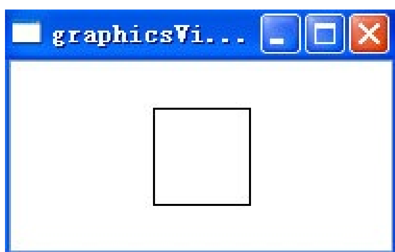
```
#include <QtGui>

int main(int argc,char* argv[ ])
{
    QApplication app(argc,argv);

    QGraphicsScene *scene = new QGraphicsScene;  //场景
    QGraphicsRectItem *item = new QGraphicsRectItem(100,100,50,50);  //矩形项
    scene->addItem(item);  //项添加到场景
    QGraphicsView *view = new QGraphicsView; //视图
    view->setScene(scene);  //视图关联场景
    view->show();  //显示视图

    return app.exec();
}
```

这里我们建立了一个最简单的基于这个图形视图框架的程序。分别新建了一个场景，一个图形项和一个视图，并将图形项添加到场景中，将视图与场景关联，最后显示视图就可以了。基于这个框架的所有程序都是这样实现的。运行效果如下。



就像我们看到的，场景是管理图形项的，所有的图形项必须添加到一个场景中，但是场景本身无法可视化，我们要想看到场景上的内容，必须使用视图。下面我们分别对图形项、场景和视图进行介绍。

二、图形项（ `QGraphicsItem` ）

`QGraphicsItem` 类是所有图形项的基类。图形视图框架对一些典型的形状提供了一些标准的图形项。比如上面我们使用的矩形（ `QGraphicsRectItem` ）、椭圆（ `QGraphicsEllipseItem` ）、文本（ `QGraphicsTextItem` ）等多个图形项。但只有继承 `QGraphicsItem` 类实现我们自定义的图形项时，才能显示出这个类的强大。 `QGraphicsItem` 支持以下功能：

- 鼠标的按下、移动、释放和双击事件，也支持鼠标悬停、滚轮和右键菜单事件。
- 键盘输入焦点和键盘事件
- 拖放
- 利用QGraphicsItemGroup进行分组
- 碰撞检测

（一）自定义图形项

1．在前面的项目中添加新的C++类，类名设为 `MyItem` ，基类设为 `QGraphicsItem` 。

2．然后，我们在 `myitem.h` 文件中添加头文件 `#include <QtGui>` 。（说明： `QtGui` 模块里面包含了所有图形界面类，所以为了简便，这里只包含了该头文件，正式开发程序时不推荐这么做！）

3．再添加两个函数的声明：

```
QRectFboundingRect() const;
voidpaint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget*widget);
```

4．下面到 `myitem.cpp` 中对两个函数进行定义：

```
QRectFMyItem::boundingRect() const
{
    qreal penWidth = 1;
    return QRectF(0 - penWidth / 2, 0 -penWidth / 2,
                  20 + penWidth, 20 + penWidth);
}

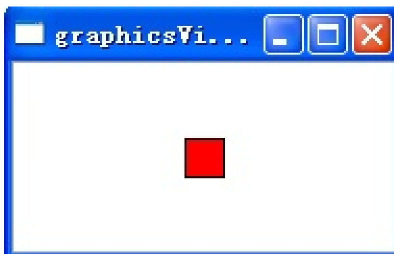

voidMyItem::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,QWidget *w
idget)
{
Q_UNUSED(option);  //标明该参数没有使用
    Q_UNUSED(widget);
    painter->setBrush(Qt::red);
    painter->drawRect(0,0,20,20);
}
```

5．下面到 `main.cpp` 中添加 `#include "myitem.h"`

然后将以前那个矩形项的定义语句改为：

```
MyItem *item =new MyItem;
```

运行程序，效果如下：



可以看到，我们要继承 `QGraphicsItem` 类实现自定义的图形项，必须先实现两个纯虚函数 `boundingRect()` 和 `paint()` ，前者用于定义 `Item` 的绘制范围，后者用于绘制图形项。其实 `boundingRect()` 还有很多用途，后面会涉及到。

（二）光标和提示

1．在 `myitem.cpp` 中的构造函数中添加两行代码，如下：

```
MyItem::MyItem()
{
    setToolTip("Click and drag me!");  //提示
    setCursor(Qt::OpenHandCursor);    //改变光标形状
}
```

然后运行程序，效果如下：



当光标放到小方块上时，光标变为了手型，并且弹出了提示。更多的光标形状可以查看 `Qt::CursorShape` ，我们也可以使用图片自定义光标形状。

（三）拖放

下面写这样一个程序，有几个不同颜色的圆形和一个大矩形，我们可以拖动圆形到矩形上，从而改变矩形的颜色为该圆形的颜色。

1．将上面的程序进行改进，用来实现圆形图形项。

在 `myitem.h` 中添加一个私有变量和几个键盘事件处理函数的声明：

```
protected:
    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
private:
    QColor color;
```

2．然后到 `myitem.cpp` 中，在构造函数中初始化颜色变量：

```
color = QColor(qrand() % 256, qrand() %256, qrand() % 256); //初始化随机颜色
```

在 `paint()` 函数中将绘制矩形的代码更改如下：

```
painter->setBrush(color);
painter->drawEllipse(0, 0, 20, 20);
```

3．下面我们定义几个键盘事件处理函数：

```
voidMyItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
   if(event->button() != Qt::LeftButton)
    {
       event->ignore();  //如果不是鼠标左键按下，则忽略该事件
       return;
    }
   setCursor(Qt::ClosedHandCursor); //如果是鼠标左键按下，改变光标形状
}

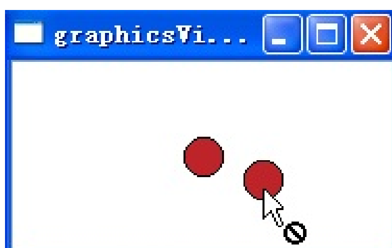

voidMyItem::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
   if(QLineF(event->screenPos(),event->buttonDownScreenPos(Qt::LeftButton))
       .length() < QApplication::startDragDistance())
{
//如果鼠标按下的点到现在的点的距离小于程序默认的拖动距离，表明没有拖动，则返回
       return;
    }
   QDrag *drag = new QDrag(event->widget()); //为event所在窗口部件新建拖动对象
   QMimeData *mime = new QMimeData; //新建QMimeData对象，它用来存储拖动的数据
   drag->setMimeData(mime); //关联
   mime->setColorData(color);  //放入颜色数据

   QPixmap pix(21,21);  //新建QPixmap对象，它用来重新绘制圆形，在拖动时显示
   pix.fill(Qt::white);
   QPainter painter(&pix);
   paint(&painter,0,0);
   drag->setPixmap(pix);

   drag->setHotSpot(QPoint(10, 15)); //我们让指针指向圆形的(10,15)点
   drag->exec();  //开始拖动
   setCursor(Qt::OpenHandCursor);   //改变光标形状
}


voidMyItem::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
   setCursor(Qt::OpenHandCursor);   //改变光标形状
}
```

此时运行程序，效果如下：



4．下面我们新添一个类，它用来提供矩形图形项，并且可以接收拖动的数据。
在 `myitem.h` 中，我们加入该类的声明：

```
class RectItem : public QGraphicsItem
{
public:
   RectItem();
   QRectF boundingRect() const;
   void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,QWidget *widge
t);
protected:
    void dragEnterEvent(QGraphicsSceneDragDropEvent *event); //拖动进入事件
    void dragLeaveEvent(QGraphicsSceneDragDropEvent *event); //拖动离开事件
    void dropEvent(QGraphicsSceneDragDropEvent *event);  //放入事件
private:
    QColor color;
    bool dragOver;  //标志是否有拖动进入
};
```

5．然后进入 `myitem.cpp` 进行相关函数的定义：

```
RectItem::RectItem()
{
   setAcceptDrops(true);  //设置接收拖放
   color = QColor(Qt::lightGray);
}

QRectF RectItem::boundingRect() const
{
    return QRectF(0, 0, 50, 50);
}

void RectItem::paint(QPainter *painter,const QStyleOptionGraphicsItem *option, QWidget
 *widget)
{
    painter->setBrush(dragOver? color.light(130) : color);  //如果其上有拖动，颜色变亮
   painter->drawRect(0,0,50,50);
}


voidRectItem::dragEnterEvent(QGraphicsSceneDragDropEvent *event)
{
   if(event->mimeData()->hasColor()) //如果拖动的数据中有颜色数据，便接收
    {
      event->setAccepted(true);
      dragOver = true;
      update();
    }
   else event->setAccepted(false);
}

voidRectItem::dragLeaveEvent(QGraphicsSceneDragDropEvent *event)
{
   Q_UNUSED(event);
   dragOver = false;
   update();
}

void RectItem::dropEvent(QGraphicsSceneDragDropEvent*event)
{
   dragOver = false;
if(event->mimeData()->hasColor())
//我们通过类型转换来获得颜色
      color =qVariantValue<QColor>(event->mimeData()->colorData());
   update();
}
```

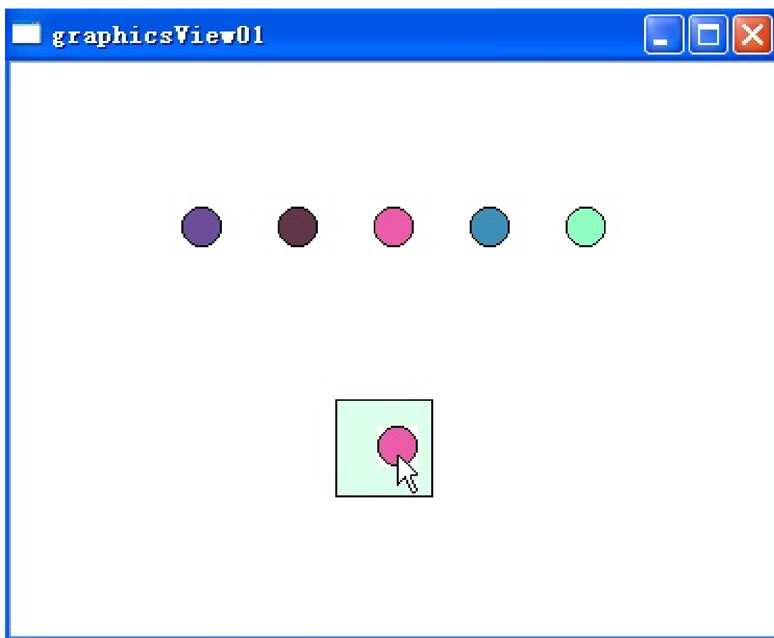
6．下面进入 `main.cpp` 文件，更改 `main()` 函数中的内容如下：

```
int main(int argc,char* argv[ ])
{
    QApplication app(argc,argv);

    qsrand(QTime(0,0,0).secsTo(QTime::currentTime())); //设置随机数初值
    QGraphicsScene *scene = new QGraphicsScene;
    for(int i=0; i<5; i++) //在不同位置新建5个圆形
     {
        MyItem *item = new MyItem;
        item->setPos(i*50+20,100);
        scene->addItem(item);
     }
    RectItem *rect = new RectItem; //新建矩形
    rect->setPos(100,200);
    scene->addItem(rect);
    QGraphicsView *view = new QGraphicsView;
    view->setScene(scene);
    view->resize(400,300); //设置视图大小
    view->show();

    return app.exec();
}
```

这是运行程序，效果如下：



这时我们已经实现了想要的效果。可以看到，要想实现拖放，必须源图形项和目标图形项都进行相关设置。在源图形项的鼠标事件中新建并执行拖动，而在目标图形项中必须指定 `setAcceptDrops(true);` 这个函数，这样才能接收拖放，然后需要实现拖放的几个事件处理函数。

（四）键盘与鼠标事件

1．新建项目 `graphicsView02`，然后按照（一）中自定义图形项进行操作（可以直接把那里的代码拷贝过来）。下面我们先来看键盘事件。

2．在 `myitem.h` 文件中声明键盘按下事件处理函数：

```
protected:
voidkeyPressEvent(QKeyEvent *event);
```

然后在 `myitem.cpp` 中进行定义：

```
void MyItem::keyPressEvent(QKeyEvent*event)
{
    moveBy(0, 10);  //相对现在的位置移动
}
```

这时运行程序，发现无论怎样方块都不会移动。其实要想使图形项接收键盘事件，就必须使其可获得焦点。我们在构造函数里添加一行代码：

```
setFlag(QGraphicsItem::ItemIsFocusable);  //图形项可获得焦点
```

（我们在新建图形项时指定也是可以的，如 `item->setFlag(QGraphicsItem::ItemIsFocusable);`）

这时运行程序，然后用鼠标点击一下方块，再按下任意按键，方块就会向下移动。效果如下图所示。



3．再看鼠标事件。我们先在 `myitem.h` 文件中声明鼠标按下事件处理函数：

```
voidmousePressEvent(QGraphicsSceneMouseEvent *event);
```

然后再 `myitem.cpp` 文件中对其进行定义：

```
voidMyItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    moveBy(10,0);
}
```

此时运行程序，点击小方块，它便会向右移动。如果我们想让鼠标可以拖动小方块，那么我们可以重新实现 `mouseMoveEvent()` 函数，还有一种更简单的方法是，我们在构造函数中指明该图形项是可移动的：

```
setFlag(QGraphicsItem::ItemIsMovable);
```

（当然我们也可以在新建图形项时指定它）

运行程序，效果如下：



（五）碰撞检测

下面先看一个例子，再进行讲解。

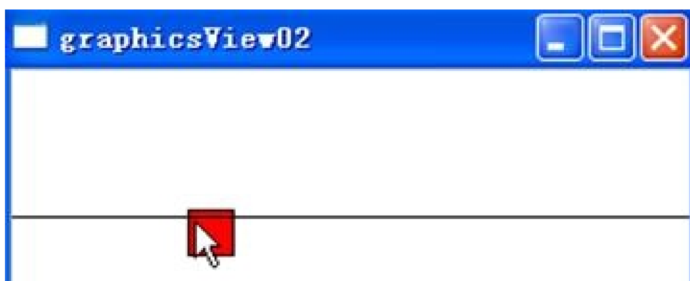
我们将上面程序中 `myitem.cpp` 文件中的 `paint()` 函数中的设置画刷的代码更改如下：

```
//如果与其他图形项碰撞则显示红色，否则显示绿色
painter->setBrush(!collidingItems().isEmpty()?Qt::red : Qt::green);
```

然后再 `main.cpp` 文件中在场景中添加一个直线图形项：

```
QGraphicsLineItem *line = newQGraphicsLineItem(0,50,300,50);
scene->addItem(line);
```

这时运行程序，效果如下：



刚开始，方块是绿色的，当我们拖动它与直线相交时，它就变成了红色。

在 `QGraphicsItem` 类中有三个碰撞检测函数，分别是 `collidesWithItem( )`、`collidesWithPath()` 和 `collidingItems()`，我们使用的是第三个。第一个是该图形项是否与指定的图形项碰撞，第二个是该图形项是否与指定的路径碰撞，第三个是返回所有与该图形项碰撞的图形项的列表。在帮助中我们可以查看它们的函数原型和介绍，这里想说明的是，这三个函数都有一个共同的参数 `Qt::ItemSelectionMode`，它指明了怎样去检测碰撞。我们在帮助中进行查看，可以发现它是一个枚举变量，一共有四个值，分别是：

- `Qt::ContainsItemShape`：只有图形项的 `shape` 被完全包含时；
- `Qt::IntersectsItemShape`：当图形项的 `shape` 被完全包含时，或者图形项与其边界相交；

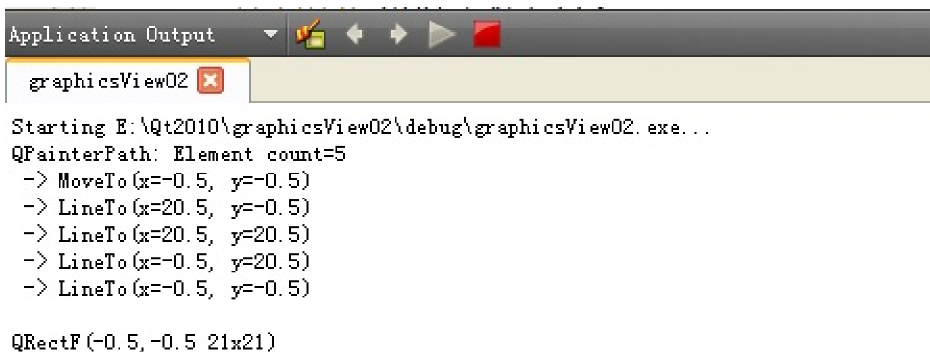
- `Qt::ContainsItemBoundingRect` ：只有图形项的 `bounding rectangle` 被完全包含时；
- `Qt::IntersectsItemBoundingRect` ：图形项的 `boundingrectangle` 被完全包含时，或者图形项与其边界相交。

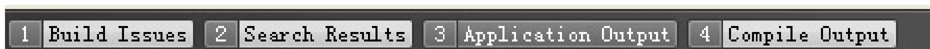
如果我们不设置该参数，那么他默认使用 `Qt::IntersectsItemShape` 。这里所说的 `shape` 是指什么呢？在 `QGraphicsItem` 类中我们可以找到 `shape()` 函数，它返回的是一个 `QPainterPath` 对象，也就是说它能确定我们图形项的形状。但是默认的，它只是返回 `boundingRect()` 函数返回的矩形的形状。下面我们具体验证一下。

在 `main.cpp` 函数中添加两行代码：

```
qDebug()<< item->shape();    //输出item的shape信息
qDebug()<< item->boundingRect();  //输出item的boundingRect信息
```

这时运行程序，在下面的程序输出窗口会输出如下信息：

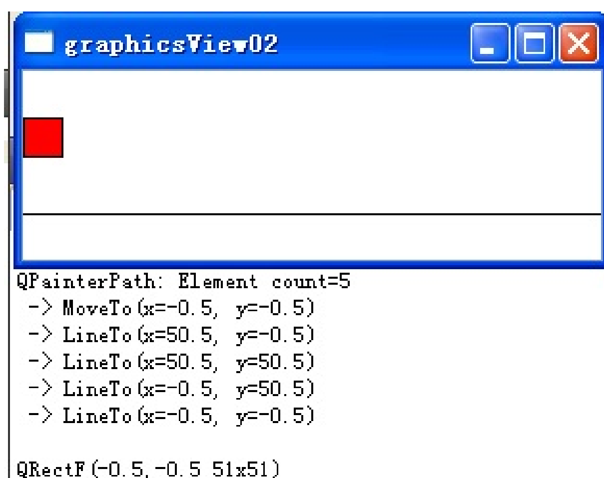




我们发现，现在 `shape` 和 `boundingRect` 的大小是一样的。这时我们在到 `myitem.cpp` 中更改函数 `boundingRect()` 函数中的内容，将大小由20，改为50：

```
return QRectF(0 - penWidth / 2, 0 -penWidth / 2,
              50 + penWidth, 50 + penWidth);
```

这时再次运行程序，效果如下：



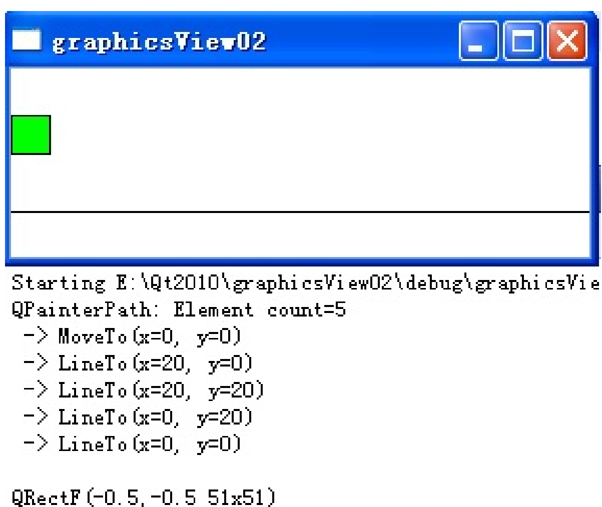
小方块一出来便成为了红色，下面的输出信息也显示了，现在 `shape` 的大小也变成了50。怎样才能使小方块按照它本身的形状，而不是其 `boundingRect` 的大小来进行碰撞检测呢？我们需要重新实现 `shape()` 函数。

在 `myitem.h` 中，我们在 `public` 里进行函数声明：`QPainterPath shape() const;`

然后到 `myitem.cpp` 中进行其定义：

```
QPainterPath MyItem::shape() const
{
    QPainterPath path;
    path.addRect(0,0,20,20);   //图形项的真实大小
    return path;
}
```

这时我们再运行程序，效果如下：



可以看到，现在 `shape` 和 `boundingRect` 的大小已经不同了。所以对于不是矩形的形状，我们都可以利用 `shape()` 函数来返回它的真实形状。

（六）移动

对于图形项的移动，我们有很多办法实现，也可以在很多层面上对其进行控制，比如说在 `View` 上控制或者在 `Scene` 上控制。但是对于大量的不同类型的图形项，怎样能一起控制呢？在图形视图框架中提供了 `advance()` 槽函数，这个函数在 `QGraphicsScene` 和 `QGraphicsItem` 中都有，在图形项类中它的原型是 `advance(int phase)` 。它的实现流程是，我们利用 `QGraphicsScene` 类的对象调用 `QGraphicsScene` 的 `advance()` 函数，这时就会执行两次该场景中所有图形项的 `advance(int phase)` 函数，第一次 `phase` 为0，告诉所有图形项即将要移动；第二次 `phase` 的值为1，这时执行移动。下面我们看一个例子。

我们在 `myitem.h` 中的 `protected` 中声明函数： `void advance(int phase);`

然后在 `myitem.cpp` 中对其进行定义：

```
void MyItem::advance(int phase)
{
    if(!phase) return;   //如果phase为0，则返回
    moveBy(0,10);
}
```

在到 `main.cpp` 中添加以下代码：

```
QTimer timer;
QObject::connect(&timer, SIGNAL(timeout()),scene, SLOT(advance()));
timer.start(1000);
```

这时运行程序，小方块就会每秒下移一下。

（七）动画

其实实现图形项的动画效果，也可以在不同的层面进行。如果我们只想控制一两个图形项的动画，一般在场景或视图中实现。但是要是想让一个图形项类的多个对象都进行同样的动画，那么我们就可以在图形项类中进行实现。我们先看一个例子。

在 `myitem.cpp` 文件中的构造函数中添加代码：

```
MyItem::MyItem()
{
    setFlag(QGraphicsItem::ItemIsFocusable); //图形项可获得焦点
    setFlag(QGraphicsItem::ItemIsMovable); //图形项可移动
    QGraphicsItemAnimation *anim = new QGraphicsItemAnimation; //新建动画类对象
    anim->setItem(this);  //将该图形项加入动画类对象中
    QTimeLine *timeLine = new QTimeLine(1000);  //新建长为1秒的时间线
    timeLine->setLoopCount(0);  //动画循环次数为0，表示无限循环
    anim->setTimeLine(timeLine); //将时间线加入动画类对象中
    anim->setRotationAt(0.5,180); //在动画时间的一半时图形项旋转180度
    anim->setRotationAt(1,360);  //在动画执行完时图形项旋转360度
    timeLine->start();  //开始动画
}
```

这时执行程序，效果如下：



小方块会在一秒内旋转一圈。我们这里使用了 `QGraphicsItemAnimation` 动画类和 `QTimeLine` 时间线类，关于这些内容我们会在后面的动画框架中细讲，所以在这里就不再介绍。

（八）右键菜单

图形项支持右键菜单，不过 `QGraphicsItem` 类并不是从 `QObject` 类继承而来的，所以它默认不能使用信号和槽机制，为了能使用信号和槽，我们需要将我们的 `MyItem` 类进行多重继承。

在 `myitem.h` 中，将 `MyItem` 类改为

```
class MyItem : public QObject, publicQGraphicsItem
{
    Q_OBJECT    //进行宏定义
    … …
}
```

这样我们就可以使用信号和槽机制了，这时我们在下面添加一个槽：

```
public slots:
   void moveTo(){setPos(0,0);}
```

因为其实现的功能很简单，我们在声明它的同时进行了定义，就是让图形项移动到 `(0,0)` 点。 然后我们在 `protected` 中声明右键菜单事件处理函数：

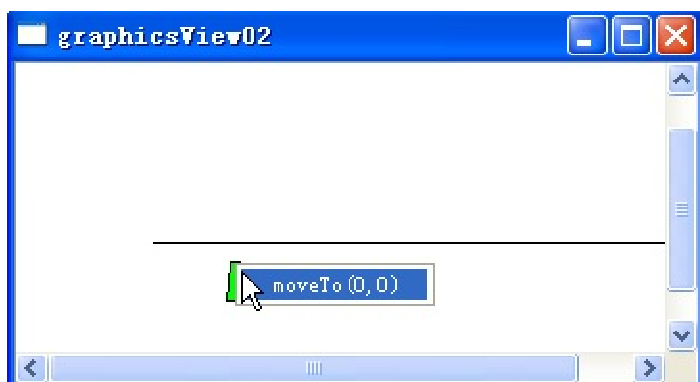
```
voidcontextMenuEvent(QGraphicsSceneContextMenuEvent *event);
```

最后我们在 `myitem.cpp` 文件中对该事件处理函数进行定义：

```
voidMyItem::contextMenuEvent(QGraphicsSceneContextMenuEvent *event)
{
   QMenu menu;
   QAction *action = menu.addAction("moveTo(0,0)");
   connect(action,SIGNAL(triggered()),this,SLOT(moveTo()));
   menu.exec(event->screenPos()); //在按下鼠标左键的地方弹出菜单
}
```

这里我们只是设置了一个菜单，当按下该菜单是，图形项移动到 `(0,0)` 点。

我们运行程序，效果如下：



结语

这一节先介绍了图形项的相关内容，而场景、视图等内容放到下一节来讲。

涉及到的源码:

- graphicsView01.zip
- graphicsView02.zip

第20篇 2D绘图（十）图形视图框架（下）

导语

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

三、场景（`QGraphicsScene`）

`QGraphicsScene`提供了图形视图框架的场景，它有以下功能：

- 提供了一个管理大量图形项的快速接口
- 向每个图形项传播事件
- 管理图形项的状态，比如选择和焦点处理
- 提供无转换的渲染功能，主要用于打印

我们新建空的Qt项目，项目名称为`graphicsView03`，完成后添加`main.cpp`文件，更改其内容如下：

```
#include <QtGui>

int main(int argc,char* argv[ ])
{
    QApplication app(argc,argv);

    QGraphicsScene scene;
    scene.addText("Hello, world!");
    QGraphicsView view(&scene);
    view.show();

    return app.exec();
}
```

运行程序，效果如下：



这里使用 `addText()` 函数添加了一个文本图形项。执行这条语句就相当于执行了下面两条语句：

```
QGraphicsTextItem*item = new QGraphicsTextItem("Hello, world!");
scene.addItem(item);
```

如果要删除一个图形项我们可以调用 `removeItem()` 函数，如：`scene.removeItem(item);`

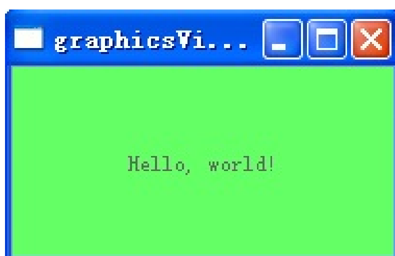
（一）场景层

一个场景分为三个层：图形项层（`ItemLayer`）、前景层（`ForegroundLayer`）和背景层（`BackgroundLayer`）。场景的绘制总是从背景层开始，然后是图形项层，最后是前景层。看下面的例子：

我们在上面的程序中添加代码：

```
scene.setForegroundBrush(QColor(255,255,255,100));
scene.setBackgroundBrush(Qt::green);
```

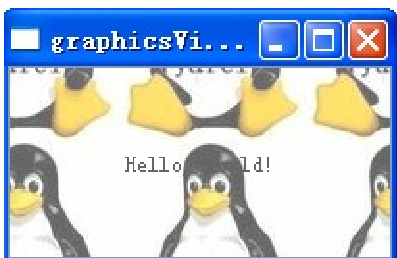
运行程序，效果如下：



对于前景层，我们一般不进行设置，或者像上面这样设置为半透明的白色。对于背景层，这里设置为了绿色，当然，我们也可以将一张图片设置为背景。

```
scene.setBackgroundBrush(QPixmap("../graphicsView03/yafeilinux.jpg"));
```

运行程序，我们可以看到，图片默认是平铺的。



如果想进一步控制前景和背景层，我们可以重新实现 `drawForeground()` 函数和 `drawBackground()` 函数。

（二）索引算法

索引算法，是指在场景中进行图形项查找的算法。`QGraphicsScene` 中提供了两种选择，它们在一个枚举变量 `QGraphicsScene::ItemIndexMethod` 中，分别是：

- `QGraphicsSecne::BspTreeIndex` ：应用Binary Space Partition tree，适合于大量的静态图形项。这个是默认值。
- `QGraphicsScene::NoIndex` ：不用索引，搜索场景中所有的图形项，适合于经常进行图形项的添加、移动和删除等操作的情况。

我们可以使用 `setItemIndexMethod()` 函数进行索引算法的更改。

（三）边界矩形

图形项可以放到场景的任何位置，场景的大小默认是没有限制的。而场景的边界矩形仅用于场景内部进行索引的维护。因为如果没有边界矩形，场景就要搜索所有的图形项，然后确定出其边界，这是十分费时的。所以如果要操作一个较大的场景，我们应该给出它的边界矩形。设置边界矩形，可以使用 `setSceneRect()` 函数。

（四）图形项查找

场景最大的优势之一就是可以快速的锁定图形项的位置，即使有上百万个图形项， `items()` 函数也能在数毫秒的时间内锁定一个图形项的位置。 `items()` 函数有几个重载函数来方便的进行图形项的查找。但是有时在场景的一个点可能重叠着几个图形项，这时我们可以使用 `itemAt()` 函数返回最上面的一个图形项。对于这些函数的使用，我们到后面讲视图时再举例讲解。

（五）事件处理和传播

场景可以传播来自视图的事件，将事件传播给该点最顶层的图形项。但是就像我们在讲图形项时所说的那样，如果一个图形项要接收键盘事件，那么它必须获得焦点。而且，如果我们在场景中重写了事件处理函数，那么在该函数的最后，必须调用场景默认的事件处理函数，只有这样，图形项才能接收到该事件。这一点我们也到后面讲视图时再细讲。

（六）打印

该部分内容也放到后面和视图一起讲。

四、视图（QGraphicsView）

`QGraphicsView` 提供了视图窗口部件，它使场景的内容可视化。你可以给一个场景关联多个视图，从而给一个数据集提供多个视口。视图部件是一个滚动区域，就是说，它可以提供一个滚动条来显示大型的场景。如果要使用OpenGL，你可以使用 `QGraphicsView::setViewport()` 函数来添加 `QGLWidget` 。

（一）缩放与旋转

我们新建空的Qt项目，项目名称为 `graphicsView04` ，然后添加 `main.cpp` 文件，再新添一个C++ 类，类名为 `MyView` ，基类为 `QGraphicsView` ，类型信息选择“继承自 `QWidget` ”。

然后在 `myview.h` 中添加头文件： `#include <QtGui>`

然后声明事件槽函数：

```
protected:
    void wheelEvent(QWheelEvent *event);
voidmousePressEvent(QMouseEvent *event);
```

我们到 `myview.cpp` 文件中进行函数的定义：

```
MyView::MyView(QWidget *parent) :
   QGraphicsView(parent)
{
   resize(400,400);
    setBackgroundBrush(QPixmap("../graphicsView04/01.jpg"));//其实就是设置场景的背景
   QGraphicsScene *scene = new QGraphicsScene(this);
   scene->setSceneRect(0,0,200,200);
   QGraphicsRectItem *item = new QGraphicsRectItem(0,0,20,20);
   item->setBrush(Qt::red);
   scene->addItem(item);
   setScene(scene);
}


void MyView::wheelEvent(QWheelEvent*event)  //滚轮事件
{
   if(event->delta() > 0)  //如果鼠标滚轮远离使用者，则delta()返回正值
       scale(0.5,0.5);  //视图缩放
   else scale(2,2);
}


void MyView::mousePressEvent(QMouseEvent*event)
{
   rotate(90);  //视图旋转顺时针90度
}
```

这里我们定义了鼠标的滚轮事件和按下事件，在滚轮事件中，利用 `delta()` 函数返回值的正负来判断滚轮的移动方向，然后我们让视图进行缩放。

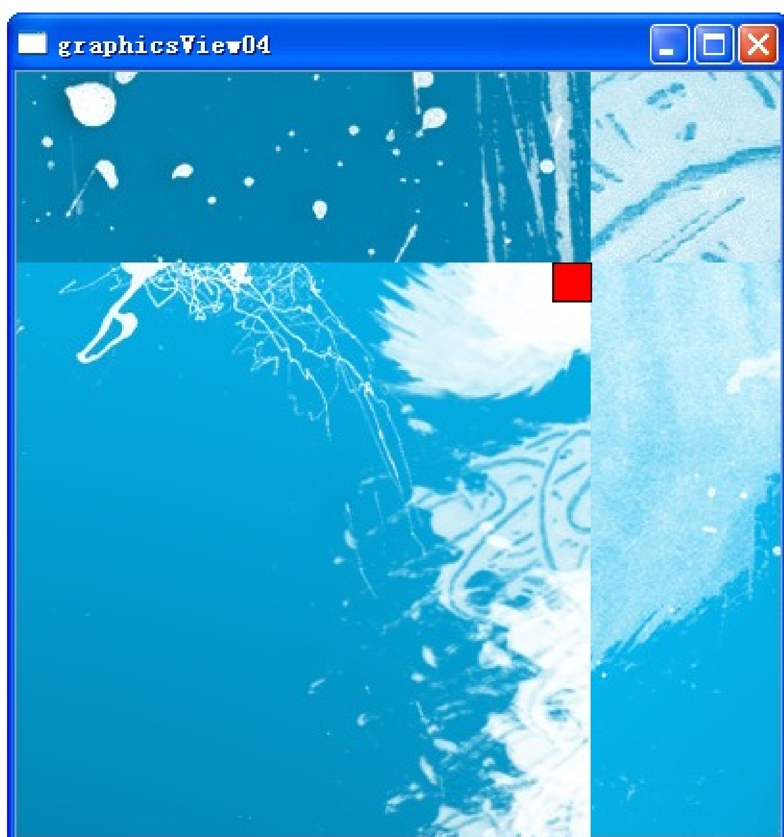
最后到 `main.cpp` 文件中，更改其内容如下：

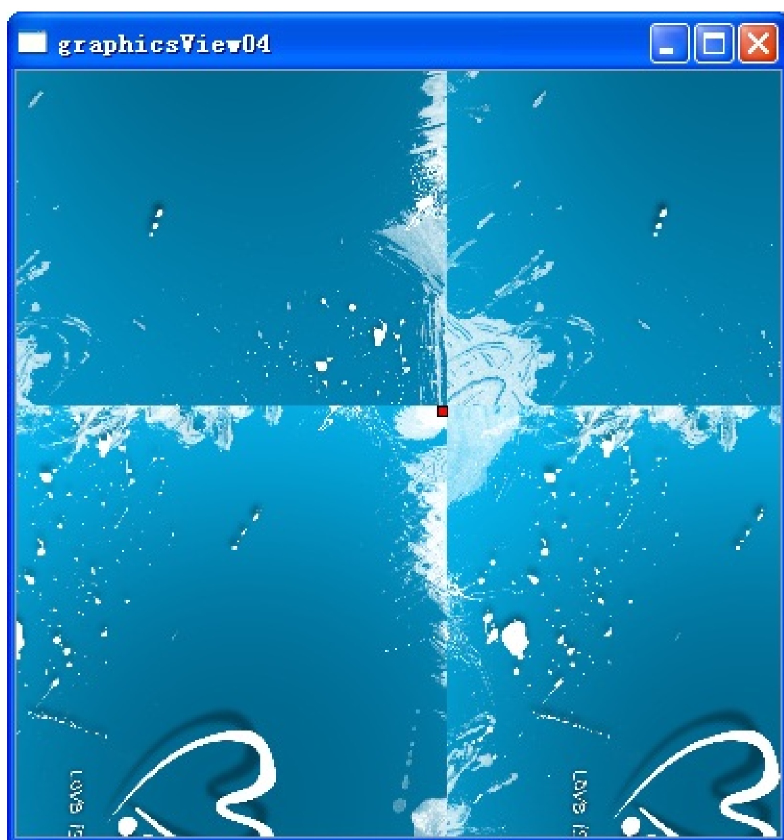
```
#include "myview.h"

int main(int argc,char *argv[])
{
   QApplication app(argc,argv);
   MyView *view = new MyView;
   view->show();
   return app.exec();
}
```

我们运行程序，效果如下：


graphicsView04


graphicsView04





上面四幅图分别是：正常，旋转90度后，缩小后，放大后的效果。可以看到实现视图的变换是十分简单的。

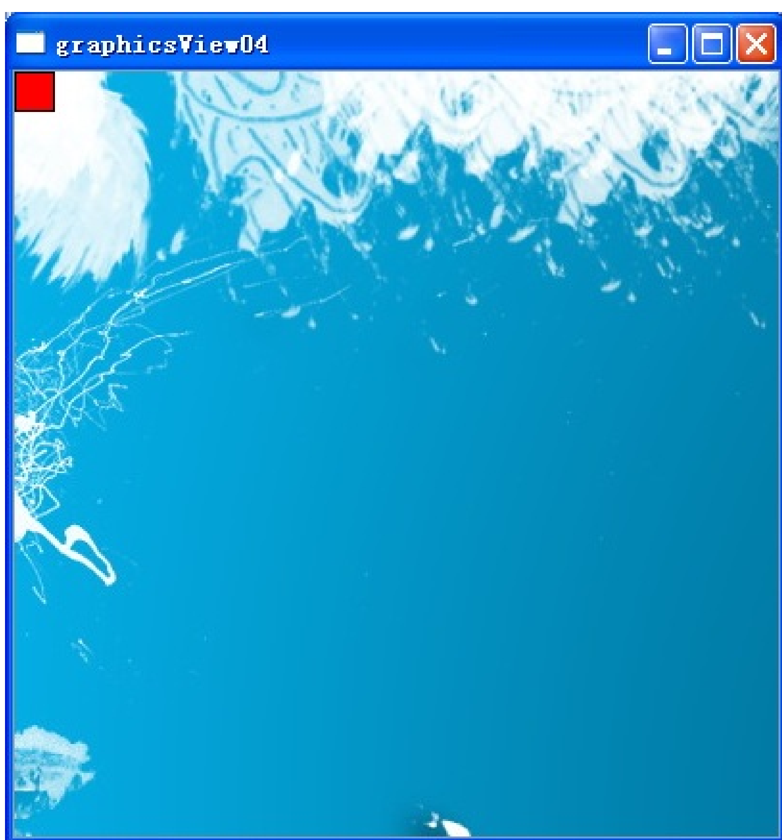
（二）场景边框与场景对齐方式

我们在上面讲场景时就提到了场景边框(`SceneRect`)，这里再说说它在视图中的作用。我们前面说过，视图是可以提供滚动条的，但是，这只是在视图窗口小于场景时才自动出现的。如果我们不定义场景边框，那么当场景中的图形项移动到视图可视窗口以外的地方时，视图就会自动出现滚动条，但是即使是图形项再次回到可视区域，滚动条也不会消失。为了解决这个问题，我们可以为场景设置边框，这样，当图形项移动到场景边框以外时，视图是不会提供额外的滚动区域的。

而当整个场景都可视时，也就是说视图没有滚动条时，我们可以通过 `setAlignment()` 函数来设置场景在视图中的对齐方式，如左对齐 `Qt::AlignLeft` ，向上对齐 `Qt::AlignTop` ，中心对齐 `Qt::AlignCenter` 。更多的对齐方式，可以查看帮助中 `Qt::Alignment` 关键字。默认的对齐方式是 `Qt::AlignCenter` 。而且几种对齐方式可以通过"按位或"操作一起使用。我们在上面的程序中的 `myitem.cpp` 文件中的构造函数最后添加一行代码：

```
setAlignment(Qt::AlignLeft | Qt::AlignTop);
```

运行效果如下图所示。



（三）拖动模式

在QGraphicView中提供了三种拖动模式，分别是：

- `QGraphicsView::NoDrag` ：忽略鼠标事件，不可以拖动。
- `QGraphicsView::ScrollHandDrag` ：光标变为手型，可以拖动场景进行移动。
- `QGraphicsView::RubberBandDrag` ：使用橡皮筋效果，进行区域选择，可以选中一个区域

内的所有图形项。

我们可以利用 `setDragMode()` 函数进行相应设置。

下面我们更改上面的程序。在 `myview.cpp` 中的构造函数中的最后添加一行代码：

```
setDragMode(QGraphicsView::ScrollHandDrag);//手型拖动
```

并将场景外框放大一点：

```
scene->setSceneRect(0,0,800,800);
```

这时运行程序，虽然出现了小手，但是并不能拖动场景。为什么呢？我们在 `mousePressEvent()` 函数中添加一行代码：

```
QGraphicsView::mousePressEvent(event);
```

这时再运行程序，发现已经成功了。效果如下：



我们在事件函数的最后添加了一行：`QGraphicsView::mousePressEvent(event);` 这样程序才能执行默认的事件。这也是我们下面要说的事件传播的内容的一部分。

（四）事件传递

在上面我们看到必须在事件函数的最后将 `event` 参数传递出去，才能执行默认的事件操作。其实不止上面那一种情况，在图形视图框架中，鼠标键盘等事件是从视图进入的，视图将它们传递给场景，场景再将事件传递给该点的图形项，如果该点有多个图形项，那么就传给最上面的图形项。所以要想使这个事件能一直传播下去，我们就需要在重新实现事件处理函数时，在其最后将 `event` 参数传给默认的事件处理函数。比如我们重写了场景的键盘按下事件处理函数，那么我们就在该函数的最后写上 `QGraphicsScene::keyPressEvent(event);` 一行代码。

（五）背景缓存

如果场景的背景需要大量耗时的渲染，可以利用 `CacheBackground` 来缓存背景，当下次需要渲染背景时，可以快速进行渲染。它的原理就是，把整个视口先绘制到一个 `pixmap` 上。但是这个只适合较小的视口，也就是说，如果视图窗口很大，而且有滚动条，那么就不再适合缓存背景。我们可以使用 `setCacheMode(QGraphicsView::CacheBackground);` 来设置背景缓存。默认设置是没有缓存 `QGraphicsView::CacheNone` 。

（六）OpenGL渲染

`QGraphicsView` 默认使用一个 `QWidget` 作为视口部件，如果我们要使用OpenGL进行渲染，可以使用 `setViewport()` 函数来添加一个 `QGLWidget` 对象。看下面的例子。

我们先在项目文件 `graphicsView04.pro` 中加入

```
QT += opengl
```

说明要使用OpenGL模块，然后在 `myview.cpp` 文件中添加头文件：

```
#include <QtOpenGL>
```

最后在构造函数中加入代码：

```
QGLWidget *widget =new QGLWidget(this);
setViewport(widget);
```

这样便使用OpenGL进行渲染了。关于OpenGL，我们在后面的3D绘图部分再讲。

（七）图形项查找与图形项组

在前面讲场景时，我们就涉及了图形项查找的内容，当时没有细讲，现在我们把它和图形项组放到一起来讲解。先看一个例子，然后再介绍。

在 `myview.cpp` 中的构造函数里将以前那个 `item` 改名为 `item1` ，然后再加入一个 `item2` 和一个图形项组对象 `group` 。更改后构造函数的部分代码如下：

```
QGraphicsRectItem *item1 = newQGraphicsRectItem(0,0,20,20);
item1->setBrush(Qt::red);
item1->setPos(10,0);
scene->addItem(item1);

QGraphicsRectItem *item2 = newQGraphicsRectItem(0,0,20,20);
item2->setBrush(Qt::green);
item2->setPos(30,0);
scene->addItem(item2);

QGraphicsItemGroup *group = newQGraphicsItemGroup;  //新建图形项组
group->addToGroup(item1);
group->addToGroup(item2);
scene->addItem(group);

setScene(scene);
qDebug() << "itemAt(10,0) :" <<itemAt(10,0); //输出(10,0)点的图形项
qDebug() << "itemAt(30,0) :" <<itemAt(30,0);
qDebug() <<"##################################"; //分割线
```

然后我们到 `myview.h` 文件中 `protected` 下声明键盘按下事件槽函数：

```
void keyPressEvent(QKeyEvent *event);
```

再到 `myview.cpp` 中定义它，如下：

```
void MyView::keyPressEvent(QKeyEvent*event)
{
   qDebug() << items();  //输出场景中所有的图形项
   items().at(0)->setPos(100,0);
   items().at(1)->setPos(0,100);
   QGraphicsView::keyPressEvent(event); //执行默认的事件处理
}
```

这时运行程序，当按下键盘上任意键后，效果如下：



下面是输出框输出的信息：

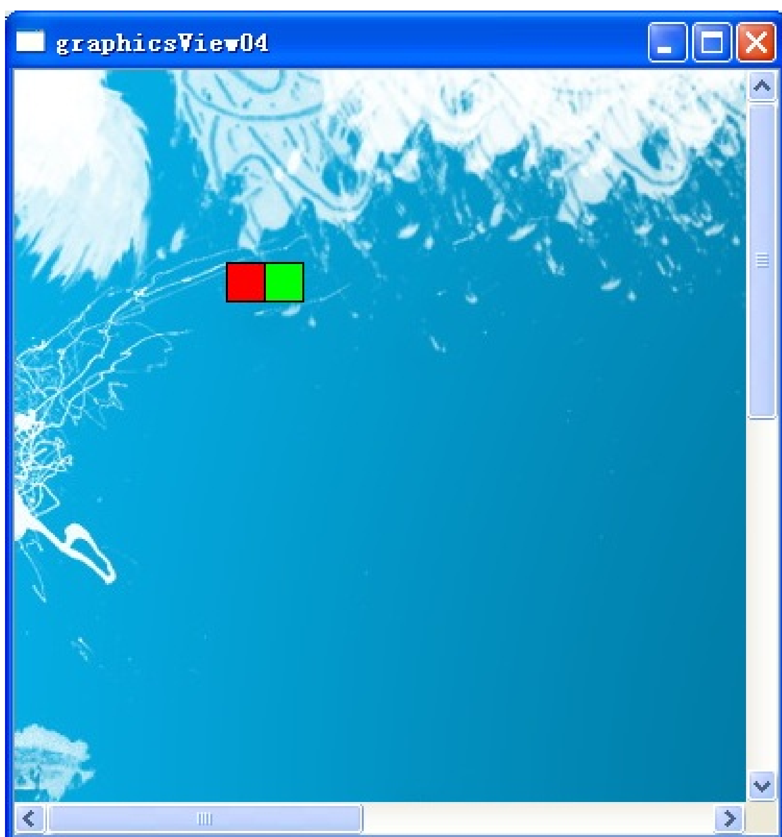
```
Application Output
graphicsView04
Starting E:\Qt2010\graphicsView04\debug\graphicsView04.exe...
itemAt(10,0) :  QGraphicsItem (this = 0x9954e08 , parent = 0x9955968 , pos = QPointF(10, 0), z =0, flags = ())
itemAt(30,0) :  QGraphicsItem (this = 0x9954148 , parent = 0x9955968 , pos = QPointF(30, 0), z =0, flags = ())
####################################
 (QGraphicsItem(this =0x9954148, parent =0x9955968, pos =QPointF(30, 0), z =0, flags = ()), QGraphicsItem(this
=0x9954e08, parent =0x9955968, pos =QPointF(10, 0), z =0, flags = ()), QGraphicsItem(this =0x9955968, parent =0x0,
pos =QPointF(0, 0), z =0, flags = ()))
```

可以看到，`itemAt()` 函数可以输出场景上任意点的图形项。而 `items()` 函数可以输出场景上所有的图形项。这里应该说明，`items()` 函数返回的图形项列表是按栈的降序排序的，也就是说，`items().at(0)` 返回的是最后加入场景的图形项。从上面可以看出，最后加入的图形项是 `item2` ，其实，因为我们使用了 `group` ，而 `item1` 和 `item2` 都在 `group` 里，所以我们只需将 `group` 加入场景中就可以了，前面把 `item1` 和 `item2` 也加入场景是多余的。我们可以将 `scene->addItem(item1);` 和 `scene->addItem(item2);` 两行代码删掉。那么这时加入场景的顺序就是，先加入 `group` ，因为 `item1` 先加入 `group` ，所以下面将 `item1` 加入场景，最后加入场景的是 `item2` ，这就是为什么 `items.at(0)` 会是 `item2` 的原因。

下面再说图形项组，其实图形项组也是一个图形项，它有图形项所拥有的所有特性。其作用就是，将加入它的所有图形项作为一个整体，对这个图形项组进行操作，就相当于对齐中所有图形项进行操作。图形项组是加入它的所有图形项的父图形项，在上面的输出的 `parent` 信息中我们可以看到这一点。下面我们将程序中的代码更改如下：

```
void MyView::keyPressEvent(QKeyEvent*event)
{
    items().at(2)->setPos(100,100);
    QGraphicsView::keyPressEvent(event); //执行默认的事件处理
}
```

运行程序，按下键盘上任意键，效果如下：



可以看到，两个图形项是同时移动的。我们要从图形项组中移除一个图形项，可以使用 `removeFromGroup()` 函数，它可以将给定的 `item` 从 `group` 中删除，要注意这时 `item` 依然存在，它会回到 `group` 的父图形项中，如果 `group` 没有父图形项，那么 `item` 就会回到场景中。我们可以使用场景的 `removeItme()` 函数来删除 `group` ，这样也会将 `group` 中所有的图形项从场景中删除。还有一种办法是利用场景的 `destroyItemGroup()` 函数，它会删除 `group` 并销毁它，但是 `group` 中的所有图形项会回到 `group` 的父图形项中，如果它没有父图形项，那么所有图形项就会回到场景中。

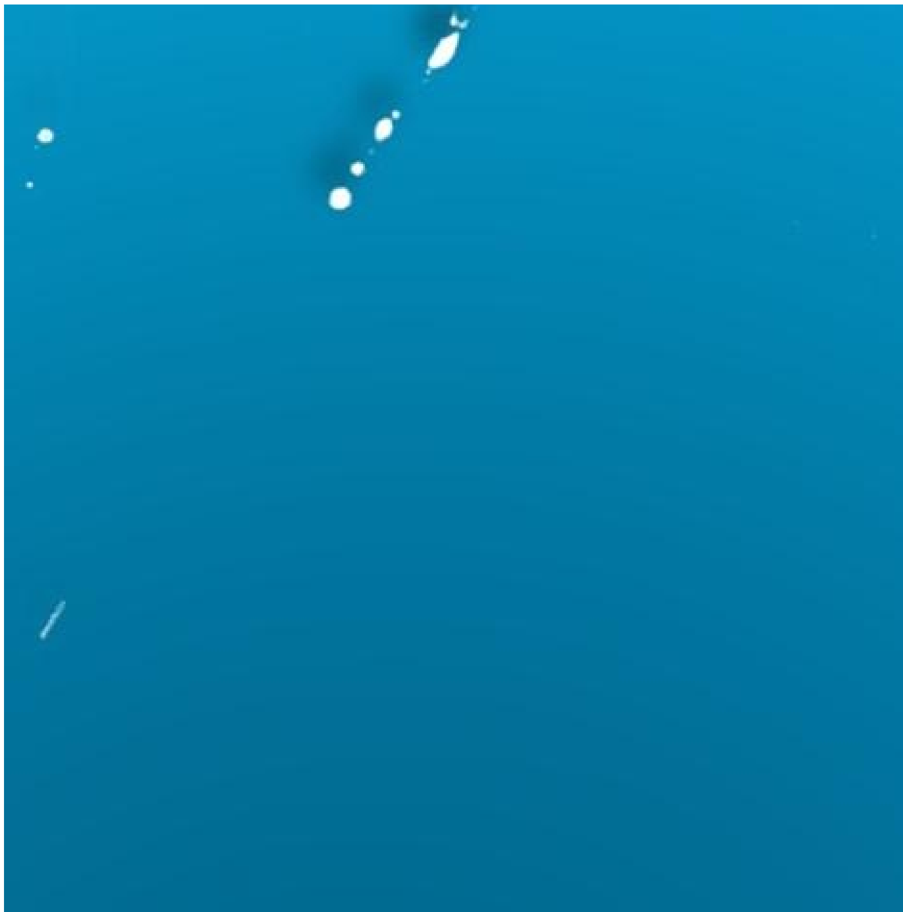
（八）打印

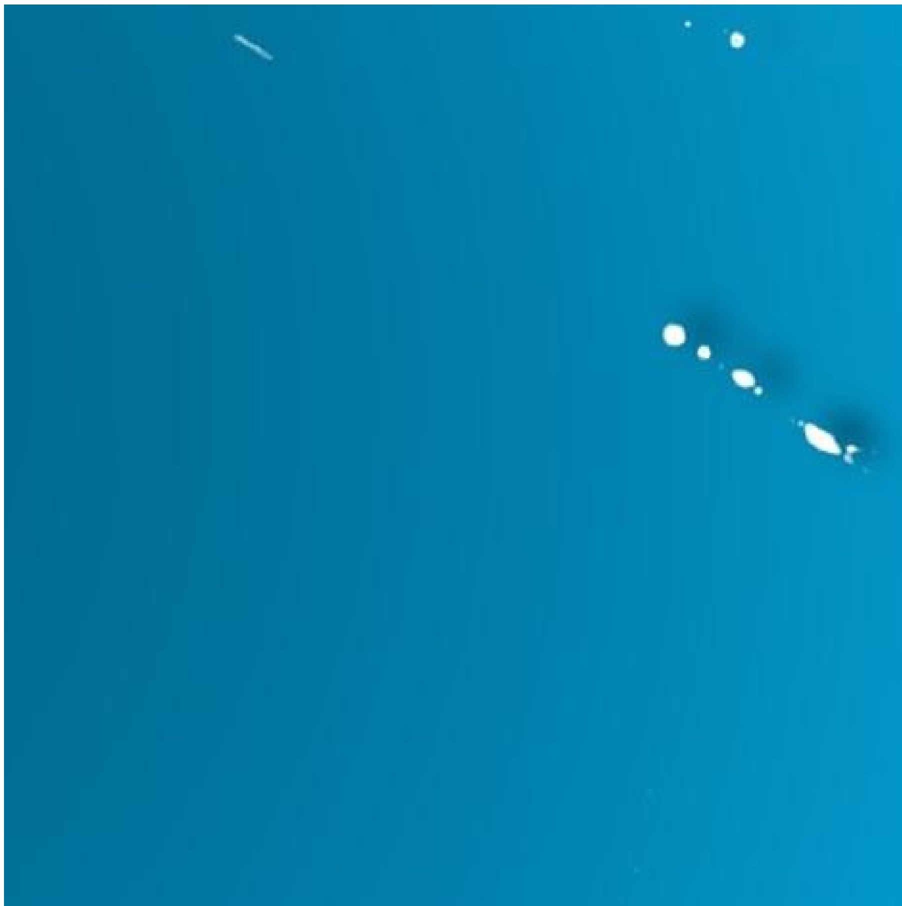
图形视图框架提供了两个打印函数 `render()` ，一个是在 `QGraphicsScene` 中，一个是在 `QGraphicsView` 中，并且它们的函数原型是一模一样的。不过它们实现的效果稍有不同。看一面的例子。

我们更改鼠标按下事件槽函数的内容如下：

```
void MyView::mousePressEvent(QMouseEvent*event)
{
     rotate(90); //视图旋转顺时针90度
    QPixmap pixmap(400,400);  //必须指定大小
    QPainter painter(&pixmap);
    render(&painter,QRectF(0,0,400,400),QRect(0,0,400,400));  //打印视图指定区域内容
     pixmap.save("../graphicsView04/save.png");
    QGraphicsView::mousePressEvent(event);
}
```

这里我们使用了视图的 `render()` 函数，其中的 `QRectF` 参数是指设备的区域，这里是指 `pixmap` 。而 `QRect` 参数是指视图上要打印的区域。我们利用 `QPixmap` 类的 `save()` 函数，将 `pixmap` 图片保存到我们项目源码目录中，文件名为 `save.png` 。下面是运行程序后，点击鼠标，生成的图片的效果：

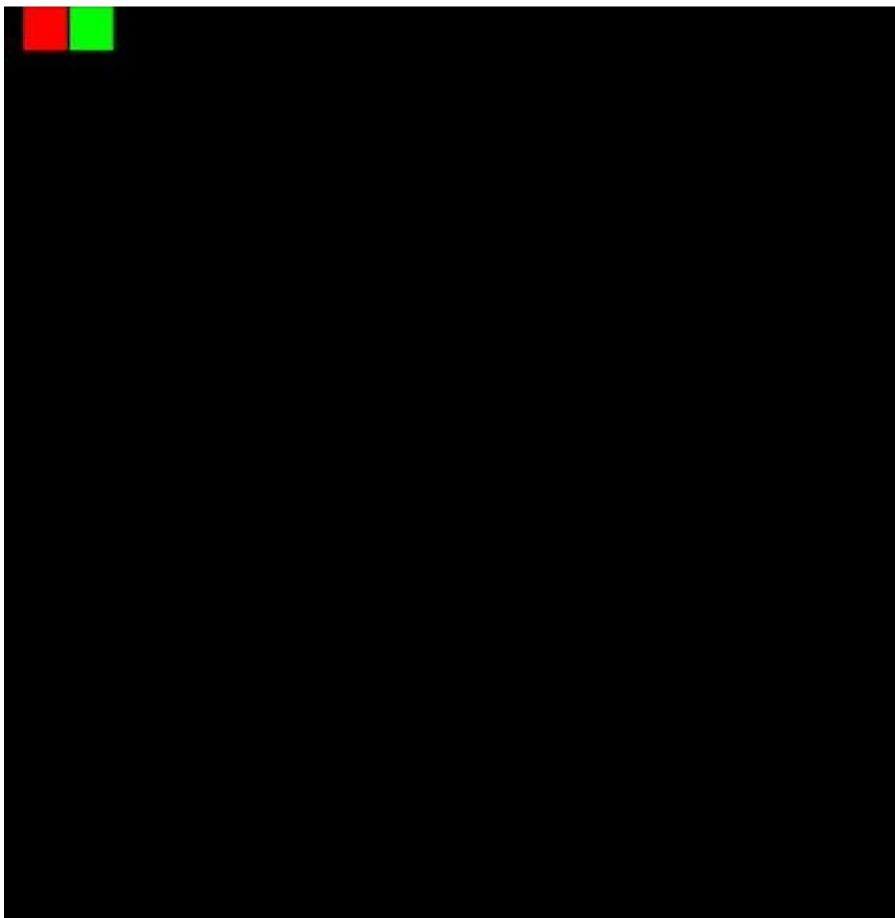




我们每点击一次鼠标，就会旋转视图，那么生成的图片就是当前视口的截图。下面我们使用场景的打印函数，将上面的打印一行的代码改为：

```
scene()->render(&painter,QRectF(0,0,400,400),QRect(0,0,400,400));//打印场景内容
```

查看图片效果：



这时无论视图怎样变换，生成的图片总是一样的。而且它并没有打印场景背景的图片。就像我们看到的，视图的打印函数是依据视图的坐标系进行打印的，我们看到的就是打印出来后的效果，它可以看做是程序窗口的截屏。而场景的打印函数，是依据场景的坐标系的，无论视图怎么转换，只要场景坐标系没有变换，它打印出来的图片都是一样的。

结语

图形视图框架是一个非常强大而且庞杂的系统，我们教程中也只是很笼统的介绍了一些最基本最常用的内容。如果大家想系统学习该部分知识，想学习如何使用该框架轻松搭建一个游戏，可以参考《Qt Creator快速入门》的第11章，以及《Qt 及Qt Quick开发实战精解》的第二章。

涉及到的源码:

- graphicsView03.zip
- graphicsView04.zip

数据篇

第21篇 数据库（一）Qt数据库应用简介

导语 下面十节讲解数据库和XML的相关内容。在学习数据库相关内容前，建议大家掌握一些基本的SQL知识，应该可以看懂基本的SELECT、INSERT、UPDATE和DELETE等语句，因为在这几篇教程中使用的都是非常简单的操作，所以即便没有数据库的专业知识也可以看懂！

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录 一、数据库简介 二、数据库驱动 三、简单的数据库应用

正文

一、数据库简介

Qt中的 `QtSql` 模块提供了对数据库的支持，该模块中的众多类基本上可以分为三层，如下图所示。

QtSql模块的类分层

用户接口层
QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel
SQL 接口层
QSqlDatabase、QSqlQuery、QSqlError、QSqlField、QSqlIndex 和 QSqlRecord
驱动层
QSqlDriver、QSqlDriverCreator<T>、QSqlDriverCreatorBase、QSqlDriverPlugin 和 QSqlResult

其中驱动层为具体的数据库和SQL接口层之间提供了底层的桥梁；SQL接口层提供了对数据库的访问，其中的 `QSqlDatabase` 类用来创建连接， `QSqlQuery` 类可以使用SQL语句来实现与数据库交互，其他几个类对该层提供了支持；用户接口层的几个类实现了将数据库中的数据链接到窗口部件上，这些类是使用前一章的模型/视图框架实现的，它们是更高层次的抽象，即便不熟悉SQL也可以操作数据库。如果要使用 `QtQql` 模块中的这些类，需要在项目文件（.pro文件）中添加 `QT += sql` 这一行代码。对应数据库部分的内容，大家可以在帮助中查看SQL Programming关键字。

二、数据库驱动

`QtSql` 模块使用数据库驱动来和不同的数据库接口进行通信。由于Qt的SQL模型的接口是独立于数据库的，所以所有数据库特定的代码都包含在了这些驱动中。Qt现在支持的数据库驱动如下图所示。

Driver Type	Description
QDB2	IBM DB2
QIBASE	Borland InterBase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTDS	Sybase Adaptive Server

需要说明的是，由于GPL许可证的兼容性问题，并不是这里列出的所有驱动插件都提供给了Qt的开源版本。下面我们通过程序来查看一下现在版本的Qt中可用的数据库插件。

1．新建Qt 控制台应用，名称为 `sqldrivers` 。

2．完成后将 `sqldrivers.pro` 项目文件中第一行代码更改为：

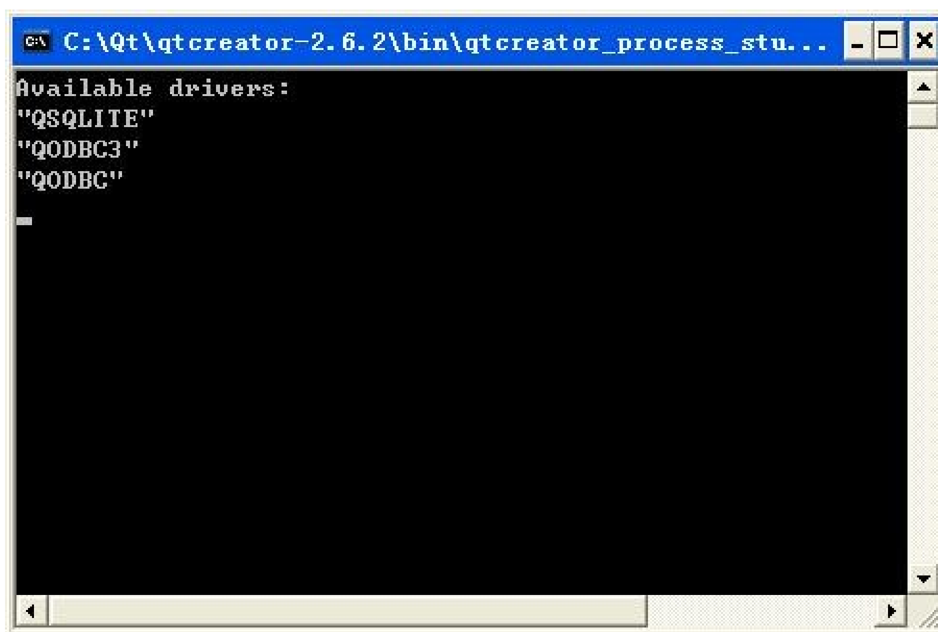
```
QT       += core sql
```

表明使用了 `sql` 模块，然后按下 `Ctrl + S` 快捷键保存该文件。

3．将 `main.cpp` 文件的内容更改如下：

```
#include <QCoreApplication>
#include <QSqlDatabase>
#include <QDebug>
#include <QStringList>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    qDebug() << "Available drivers:";
    QStringList drivers = QSqlDatabase::drivers();
    foreach(QString driver, drivers)
       qDebug() << driver;
    return a.exec();
}
```

这里先使用 `drivers()` 函数获取了现在可用的数据库驱动，然后分别进行了输出。运行程序，结果如下图所示。



可以发现，现在只支持三个数据库。这里要重点提一下SQLite数据库，它是一款轻型的文件型数据库，主要应用于嵌入式领域，支持跨平台，而且Qt对它提供了很好的默认支持，所以在本章后面的内容中，我们将使用该数据库作为例子来进行讲解。

三、简单的数据库应用

下面使用QSLite数据库来进行一个简单的演示，创建一个数据库表，然后查找其中的数据并进行输出。我们更改 `main.cpp` 文件的内容如下：

```
#include <QCoreApplication>
#include <QSqlDatabase>
#include <QDebug>
#include <QSqlQuery>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    //添加数据库驱动
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    //设置数据库名称
    db.setDatabaseName(":memory:");
    //打开数据库
    if(!db.open())
    {
        return false;
    }

    //以下执行相关sql语句
    QSqlQuery query;

    //新建student表，id设置为主键，还有一个name项
    query.exec("create table student(id int primary key,name varchar)");

    //向表中插入3条记录
    query.exec("insert into student values(1,'xiaogang')");
    query.exec("insert into student values(2,'xiaoming')");
    query.exec("insert into student values(3,'xiaohong')");

    //查找表中id >=2 的记录的id项和name项的值
    query.exec("select id,name from student where id >= 2");

    //query.next()指向查找到的第一条记录，然后每次后移一条记录
    while(query.next())
    {
        //query.value(0)是id的值，将其转换为int型
        int value0 = query.value(0).toInt();
        QString value1 = query.value(1).toString();
        //输出两个值
        qDebug() << value0 << value1 ;
    }

    return a.exec();
}
```

这里使用了SQLite数据库，数据库名为 `:memory:` 表示这是建立在内存中的数据库，也就是说该数据库只在程序运行期间有效。如果需要保存该数据库文件，我们可以将它更改为实际的文件路径。程序中使用到的 `QSqlQuery` 类，将在后面的内容中讲到。运行程序，结果如下图所示。

```
C:\Qt\qtcreator-2.6.2\bin\qtcreator_process_...
2 "xiaoming"
3 "xiaohong"
```

结语

本节简单介绍了一下Qt中数据库相关的内容，可以看到，现在Qt支持的数据库仅有两类。如何才能让Qt支持其他的数据库呢，下一节，我们将以现在广为使用的MySql为例，讲解数据库驱动的编译。如果大家想系统的学习Qt数据库部分内容，可以参考《Qt Creator快速入门》的第17章。

涉及到的源码

第22篇 数据库（二）编译MySQL数据库驱动

导语

在上一节的末尾我们已经看到，现在可用的数据库驱动只有两类3种，那么怎样使用其他的数据库呢？在Qt中，我们需要自己编译其他数据库驱动的源码，然后当做插件来使用。下面就以现在比较流行的MySQL数据库为例，说明一下怎样在QtCreator中编译数据库驱动。

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

正文

一、查看怎样编译数据库驱动

1．在Qt Creator的帮助模式索引SQL Database Drivers关键字，这篇文档里详细介绍了Qt数据库的相关内容。

2．我们在文档中定位到How to Build the QMYSQL Plugin on Windows一段，这里讲解了怎样在Windows下编译MySQL驱动。如下图所示。

How to Build the QMYSQL Plugin on Windows

You need to get the MySQL installation files. Run `SETUP.EXE` and choose "Custom Install". Install the "Libs & Include Files" Module. Build the plugin as follows (here it is assumed that MySQL is installed in `C:\MySQL`):

```
cd %QTDIR%\src\plugins\sqldrivers\mysql
qmake "INCLUDEPATH+=C:\MySQL\include" "LIBS+=C:\MYSQL\MySQL Server <version>\lib\opt\libmysql.lib"
nmake
```

If you are not using a Microsoft compiler, replace `nmake` with `make` in the line above.

可以看到，主要分为两步，首先下载并安装MySQL，在安装时要使用Custom Install定制安装，安装上库Libs和头文件Include Files；然后是编译，注意在编译驱动前先添加上MySQL的库和头文件。

二、下载MySQL

1．我们先进入MySQL的主页 http://www.mysql.com/ ，然后点击左上角的Downloads，如下图所示。



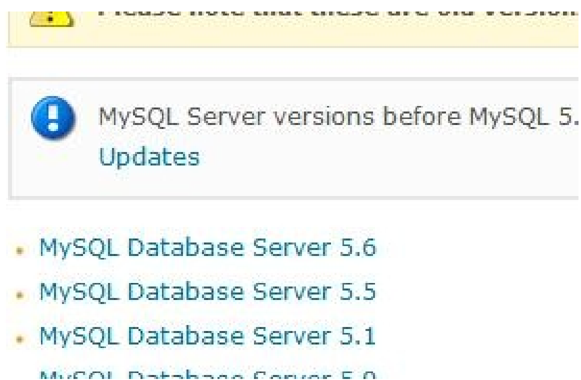
2．然后进入该页面最下面的MySQL Community Edition(GPL)链接，我们使用遵循GPL协议的开源版本。如下图所示。



3．在新的页面我们选择左上角Downloads按钮下面的Archives选项，从档案中下载。如下图所示。



4．在这个页面我们选择现在最新的MySQLDatabase Server 5.6版本。如下图所示。



5·下面我们选择按照平台分类里的MicrosoftWindows (34 files)。如下图所示。

Software Downloads by Platform

- Source and other files (98 files)
- Linux (4 files)
- SuSE Linux Enterprise Server 10 RPM (67 files)
- SuSE Linux Enterprise Server 11 RPM (100 files)
- Red Hat Enterprise Linux 4 RPM (67 files)
- Red Hat Enterprise Linux 5 RPM (142 files)
- Red Hat/Oracle Enterprise Linux 6 RPM (105 files)
- Generic Linux RPM (142 files)
- Microsoft Windows (34 files)
- Mac OS X (66 files)
- Sun Solaris (88 files)
- FreeBSD (21 files)

6·这里我们下载最新版本的第二个链接MicrosoftWindows 32. (Windows Installer format) (1 Feb 2013, 35.5M)，如下图所示。

Microsoft Windows

5.6.10

Microsoft Windows 32. (ZIP format) (23 Jan 2013, 207.3M)
Signature MD5: a8f720b353f8848ee78b49338eacf91e

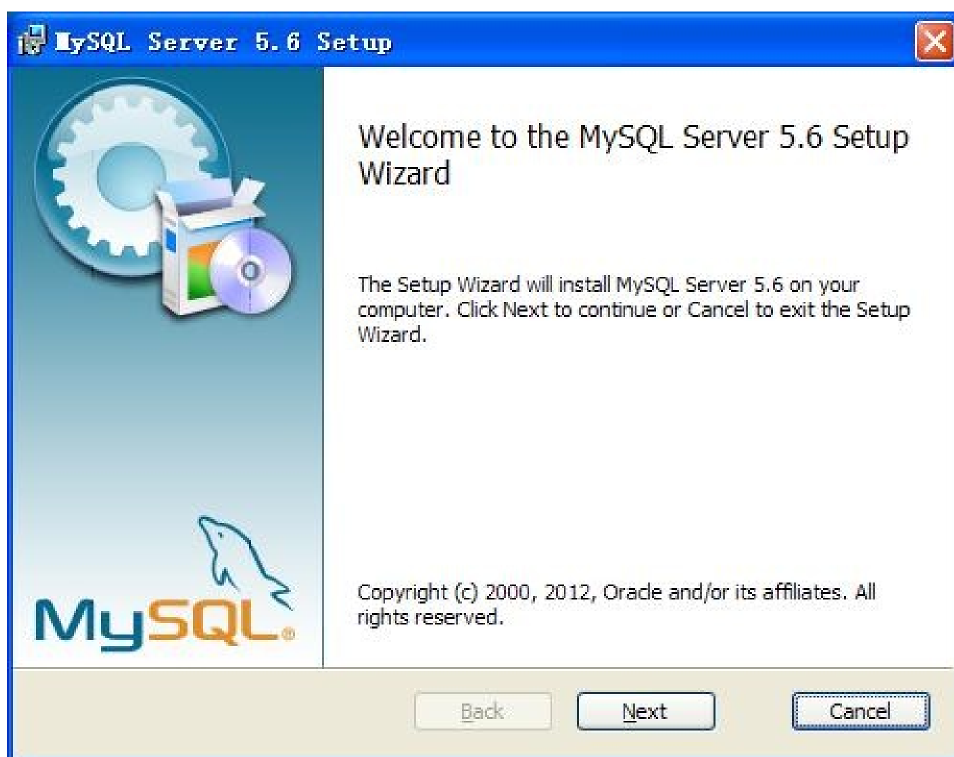
Microsoft Windows 32. (Windows Installer format) (1 Feb 2013, 35.5M)
Signature MD5: 7b76602cdf88c22677b1f29c8a516e56

Microsoft Windows (Windows Installer format) (1 Feb 2013, 37.6M)
Signature MD5: 84cc810331ebf473162a24d736986f87

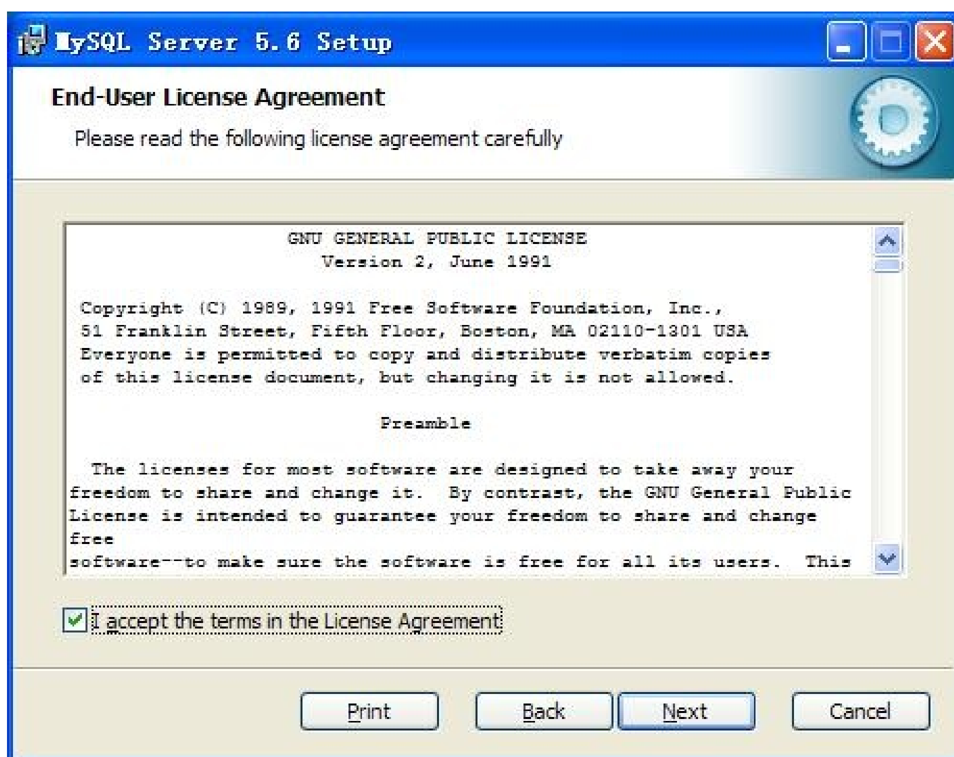
7·下载后的最终文件为：`mysql-5.6.10-win32`（大家也可以下载我们上传到网上的软件包，因为下面的操作只需要MyQSL的库、头文件以及最基本的功能，所以我们下载了该版本）

三、安装MySQL

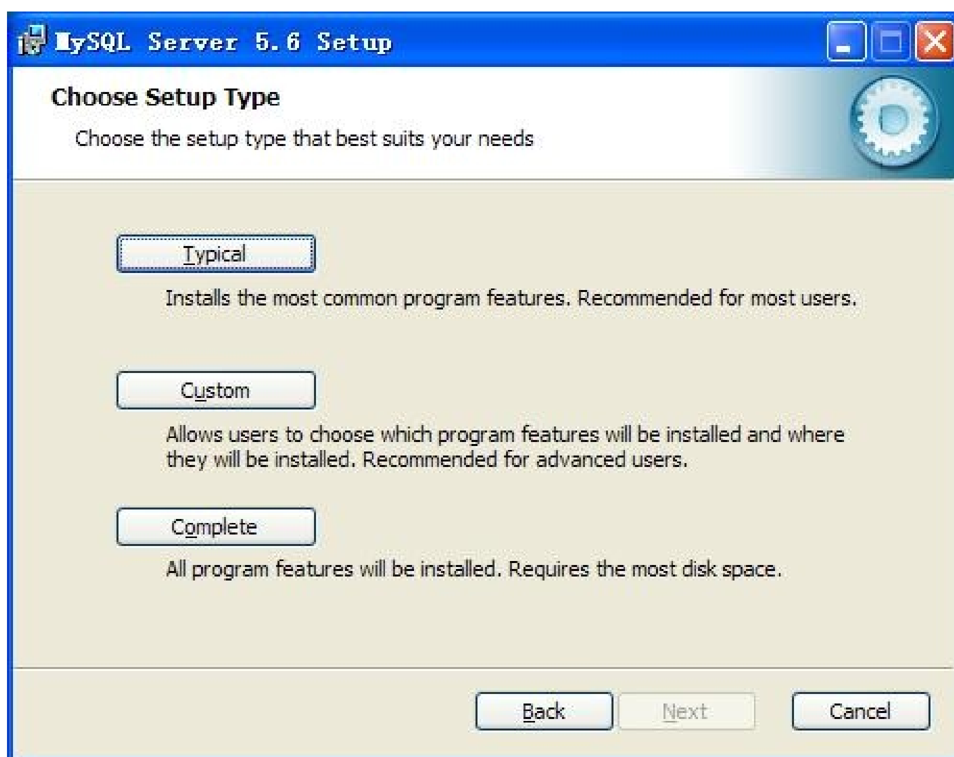
1·运行下载的安装包，如下图所示。



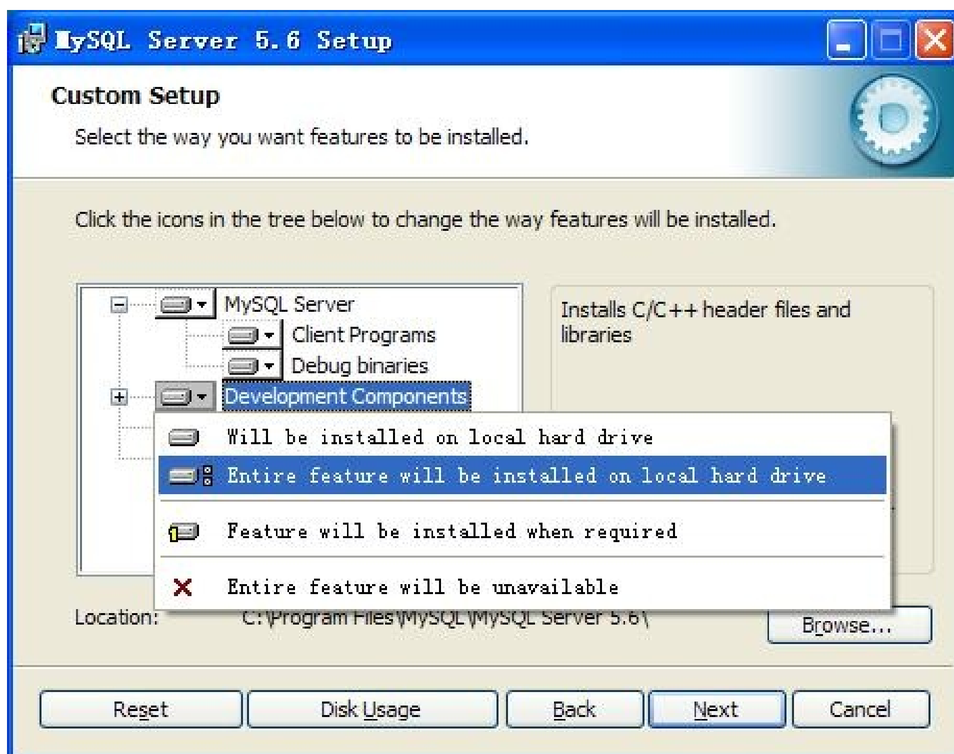
2．点击Next，进入下一步，这里我们选择同意条款。如下图所示。



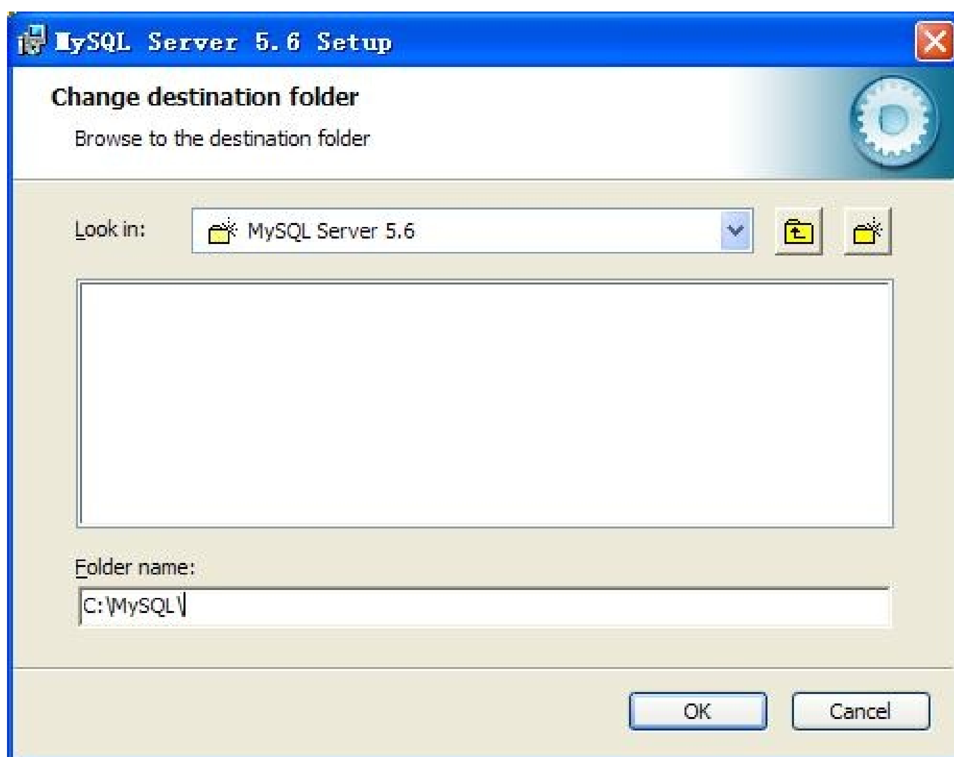
3．接着Next，下面我们选择定制安装Custom。如下图所示。



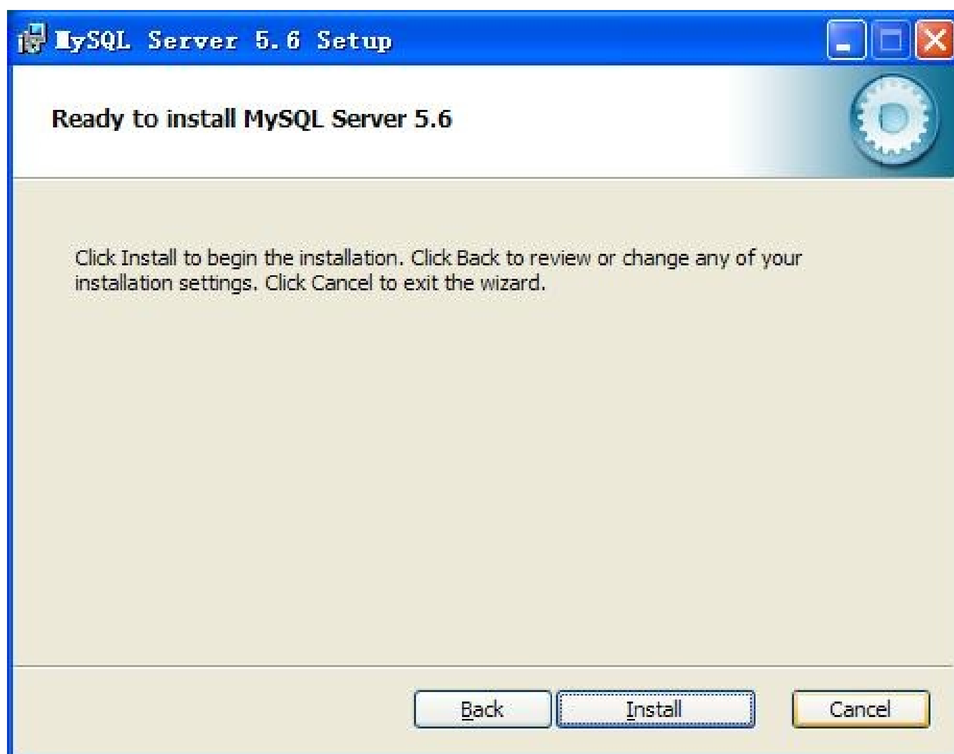
4・这里需要安装所有的头文件和库，点击Development Components前面的下拉箭头，然后选择第二项。如下图所示。大家也可以看下右边的说明。



5・然后选择下面的Browse...来更改安装路径，这里设置为 `C:\MySQL\` ，如下图所示。



6·填写完路径后点击ok回到主页面，点击Next来到新的页面，这里选择Install来进行安装。如下图所示。

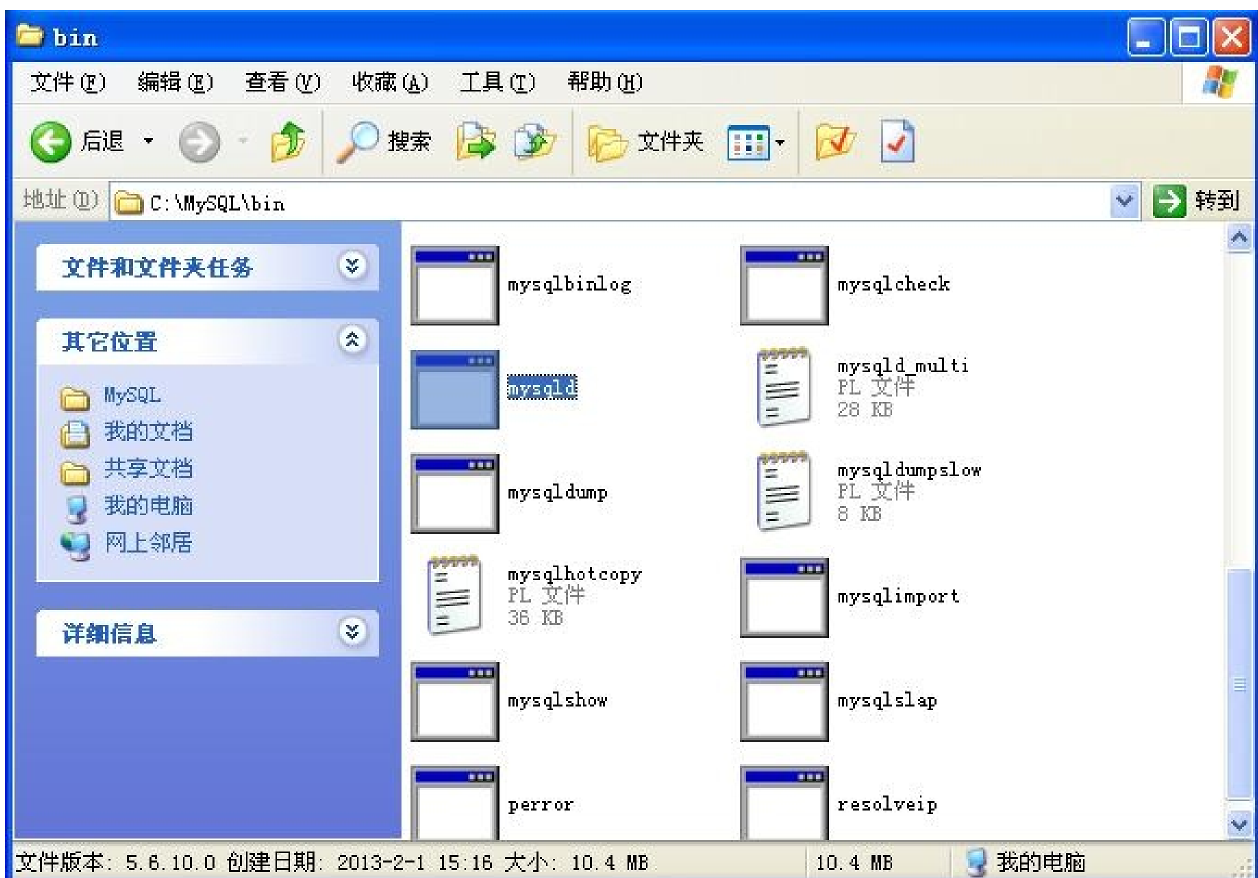


7·等安装完毕后，点击Finish完成安装。如下图所示。



四、在MySQL中创建数据库

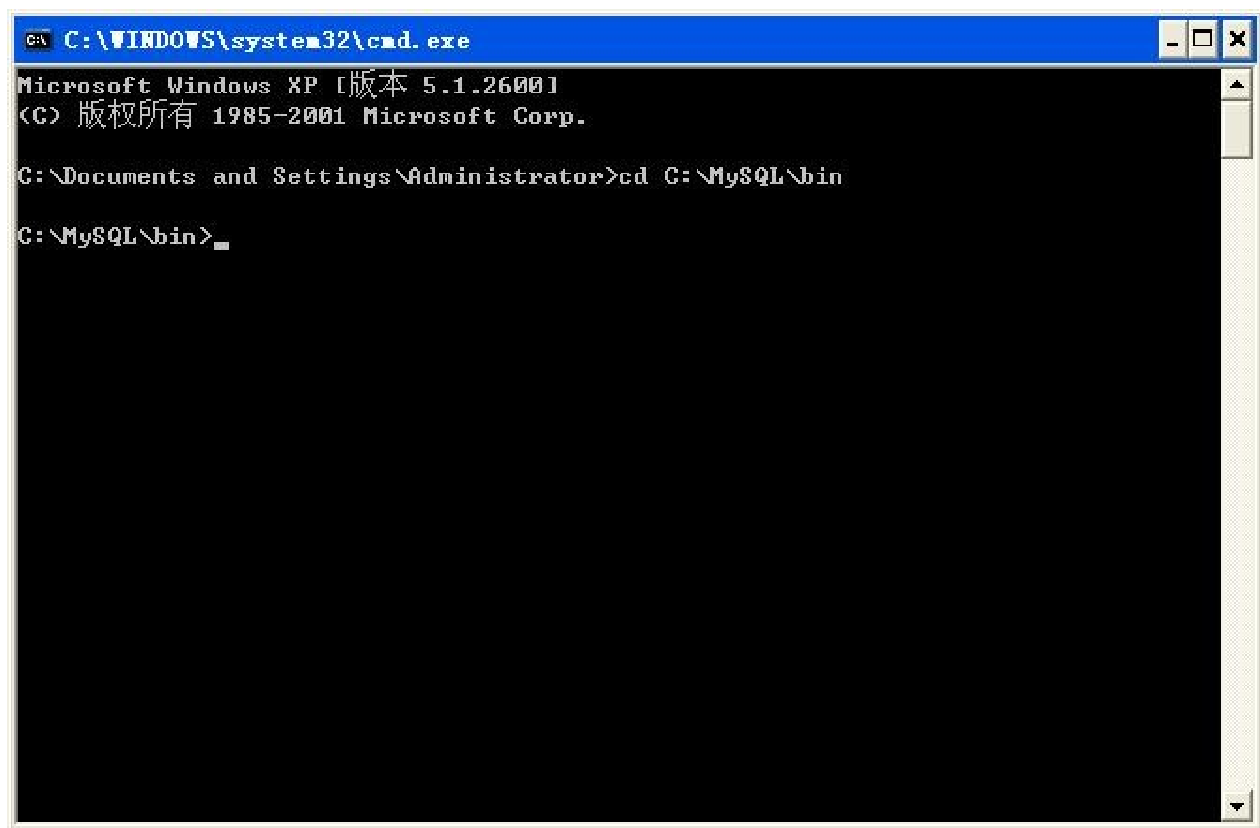
1．下面我们先在安装的MySQL中创建一个数据库，用于后面的测试。首先到MyQSL的安装目录 `C:\MySQL\bin` 目录下运行 `mysqld.exe` 程序，该程序运行完成后会自动关闭。如下图所示。



2．在Windows开始中点击“运行”，然后输入 `cmd` ，进入终端后我们输入下面的命令：

```
cd C:\MySQL\bin
```

跳转到安装目录下。如下图所示。



3．然后输入下面的命令：

```
mysql -uroot -p
```

我们使用root用户来登陆MySQL，因为默认密码是空的，所以这里不用设置密码。运行这行代码会提示Enterpassword，我们这时敲回车即可。如下图所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>cd C:\MySQL\bin

C:\MySQL\bin>mysql -uroot -p
Enter password: _
```

4·登录MySQL以后，我们使用下面的命令来查看现有的数据库：

```
show databases；
```

注意后面有个分号。如下图所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>cd C:\MySQL\bin

C:\MySQL\bin>mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.10 MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement

mysql> show databases;
```

可以看到，这里现在已经有几个数据库了，他们是MySQL需要的。如下图所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
C:\MySQL\bin>mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.6.10 MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
3 rows in set (0.00 sec)

mysql>
```

5．我们不使用已经有的数据库，而是新建自己的数据库，下面新建名为 `mydata` 的数据库：

```
create database mydata；
```

如下图所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
3 rows in set (0.00 sec)

mysql> create database mydata;
Query OK, 1 row affected (0.00 sec)

mysql> _
```

6．我们再次查看已经存在的数据库，发现显示出了刚才创建的数据库，如下图所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
3 rows in set (0.00 sec)

mysql> create database mydata;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mydata             |
| mysql              |
| performance_schema |
+--------------------+
4 rows in set (0.00 sec)

mysql>
```

7．完成后，可以输入 `exit` 退出MySQL。

这里只是简单介绍了一下在MySQL中创建数据库的基本步骤，如果大家想学习更多的MySQL的使用，请参考其他资料。

五、编译MySQL驱动

1．我们进入Qt安装目录的mysql源码目录中，具体路径为（这里Qt安装在了C盘）：

```
C:\Qt\4.8.4\src\plugins\sqldrivers\mysql
```

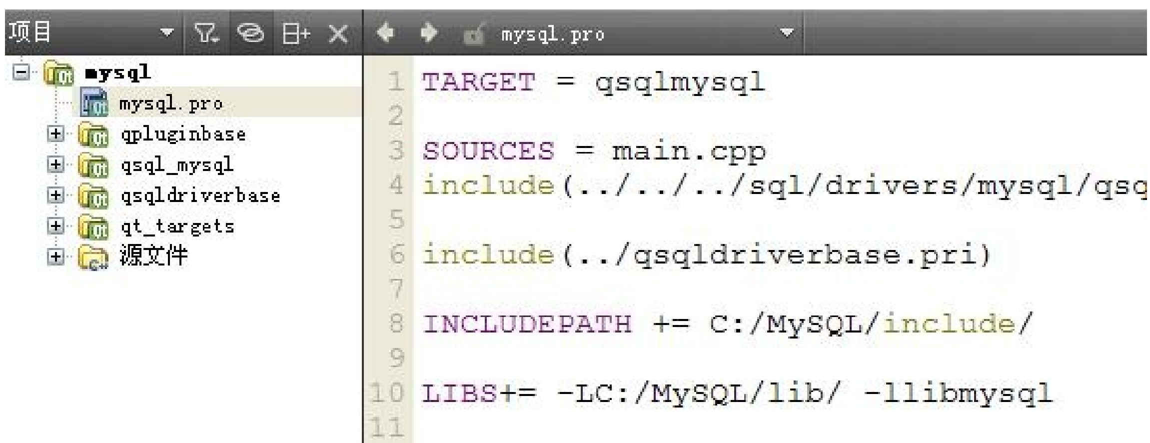
如下图所示。



2．我们使用Qt Creator打开里面的 `mysql.pro` 项目文件。然后打开 `mysql.pro` 文件，在最下面添加下面两行代码：

```
INCLUDEPATH += C:/MySQL/include/
LIBS+= -LC:/MySQL/lib/ -llibmysql
```

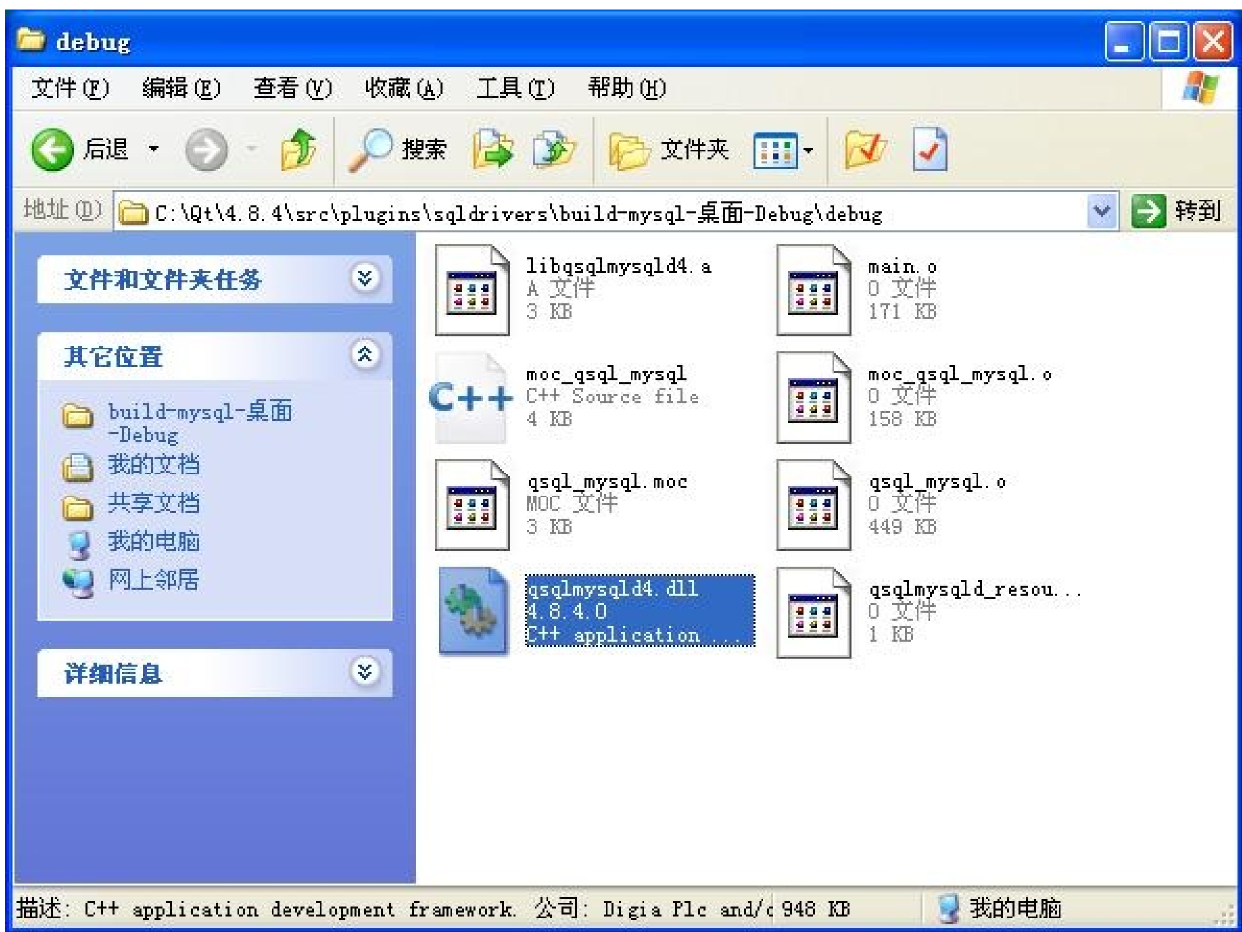
这样便包含了MySQL的库和头文件。如下图所示。



3．现在我们使用左下角的锤子 按钮来构建项目。默认构建的是Debug版本，会在Qt目录的mysql目录的同层目录里面生成构建目录，如下图所示。



里面的debug目录里有我们需要的 `qsqlmysqld4.dll` 和 `libqsqlmysqld4.a` 文件，不过它们只用于开发Debug版本的MySQL程序。如下图所示。

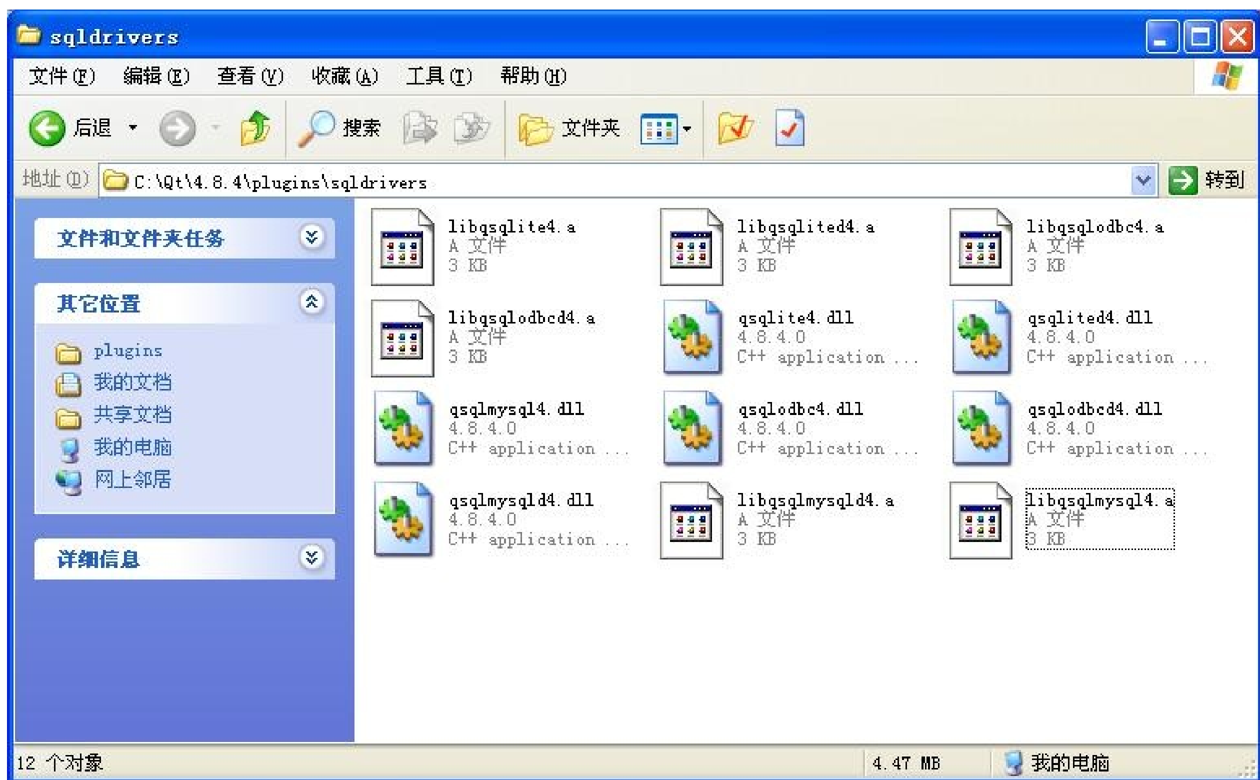


4、为了生成release库，我们在Qt Creator中运行按钮那里设置为编译Release版本。如下图所示。



5、下面再次按下锤子按钮来编译项目，就会生成`build-mysql-桌面-Release`样式的构建目录，里面包含了发布release版本程序需要的dll文件`qsqlmysql4.dll`。

6、我们将生成的`qsqlmysql4.dll`，`libqsqlmysql4.a`，`qsqlmysqld4.dll`，`libqsqlmysqld4.a`都复制到`C:\Qt\4.8.4\plugins\sqldrivers`目录下，这是数据库驱动插件放置的目录。如下图所示。



六、测试MySQL程序

1．新建Qt控制台应用，名称为`sqldriver`s。完成后在`pro`文件中更改如下：

```
QT       += core   sql
```

2．更改`main.cpp`文件内容如下。

```
#include <QCoreApplication>
#include <QSqlDatabase>
#include <QDebug>
#include <QStringList>
#include <QSqlQuery>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    // 输出可用数据库
    qDebug() << "Available drivers:";
    QStringList drivers = QSqlDatabase::drivers();
    foreach(QString driver, drivers)
        qDebug() << driver;

    // 打开MySQL
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("mydata");
    db.setUserName("root");
    db.setPassword("");
    if (!db.open())
        qDebug() << "Failed to connect to root mysql admin";
    else qDebug() << "open";

    QSqlQuery query(db);

    //注意这里varchar一定要指定长度，不然会出错
    query.exec("create table student(id int primary key,name varchar(20))");

    query.exec("insert into student values(1,'xiaogang')");
    query.exec("insert into student values(2,'xiaoming')");
    query.exec("insert into student values(3,'xiaohong')");

    query.exec("select id,name from student where id >= 2");

    while(query.next())
    {
        int value0 = query.value(0).toInt();
        QString value1 = query.value(1).toString();
        qDebug() << value0 << value1 ;
    }

    return a.exec();
}
```

这里注意，创建表时 `varchar` 一定要指定长度。现在运行程序，会出现如下图所示的结果：

```
C:\Qt\Qt5.0.2\Tools\QtCreator\bin\qtcreator_process_stub.exe
Available drivers:
"QSQLITE"
"QMYSQL3"
"QMYSQL"
"QODBC3"
"QODBC"
QSqlDatabase: QMYSQL driver not loaded
QSqlDatabase: available drivers: QSQLITE QMYSQL3 QMYSQL QODBC3 QODBC
Failed to connect to root mysql admin
QSqlQuery::exec: database not open
QSqlQuery::exec: database not open
QSqlQuery::exec: database not open
QSqlQuery::exec: database not open
QSqlQuery::exec: database not open
```

这里提示MySQL驱动没有加载。

3．我们到 `C:\MySQL\lib` 中将 `libmysql.dll` 文件复制到 `C:\Qt\4.8.4\bin` 中，然后再次运行程序，发现已经成功了，如下图所示。

```
C:\Qt\Qt5.0.2\Tools\QtCreator\bin\qtcreator_process_stub.exe
Available drivers:
"QSQLITE"
"QMYSQL3"
"QMYSQL"
"QODBC3"
"QODBC"
open
2 "xiaoming"
3 "xiaohong"
```

结语

在Qt中编译MySQL数据库驱动的内容到这里就介绍完了。从下一篇开始，我们将以SQLite数据库为范例来讲解Qt数据库部分的应用。

涉及到的代码

第23篇 数据库（三）利用 `QSqlQuery` 类执行SQL语句

导语

SQL即结构化查询语言，是关系数据库的标准语言。前面两节中已经在Qt里利用 `QSqlQuery` 类执行了SQL语句，这一节我们将详细讲解该类的使用。需要说明，因为我们重在讲解Qt中的数据库使用，而非专业的讲解数据库知识，所以不会对数据库中的一些知识进行深入讲解。

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

正文

一、创建数据库连接

前面我们是在主函数中创建数据库连接，然后打开并使用。实际中为了明了方便，一般将数据库连接单独放在一个头文件中。下面来看一个例子。

1．新建Qt Gui应用，项目名称为 `myquery` ，基类为 `QMainWindow` ，类名为 `MainWindow` 。完成后打开 `myquery.pro` 并将第一行代码更改为：

```
QT       += coregui sql
```

然后保存该文件。

2．向项目中添加新的C++头文件，名称为 `connection.` h，然后打开该文件，更改如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(":memory:");
    if (!db.open()) {
       QMessageBox::critical(0, qApp->tr("Cannot open database"),
           qApp->tr("Unable to establisha database connection."
                     ), QMessageBox::Cancel);
       return false;
    }
    QSqlQuery query;
    query.exec("create table student (id int primary key, "
               "name varchar(20))");
    query.exec("insert into student values(0, 'first')");
    query.exec("insert into student values(1, 'second')");
    query.exec("insert into student values(2, 'third')");
    query.exec("insert into student values(3, 'fourth')");
    query.exec("insert into student values(4, 'fifth')");
    return true;
}
#endif // CONNECTION_H
```

在这个头文件中我们添加了一个建立连接的函数，使用这个头文件的目的就是要简化主函数中的内容。这里先创建了一个SQLite数据库的默认连接，设置数据库名称时使用了`“:memory:”`，表明这个是建立在内存中的数据库，也就是说该数据库只在程序运行期间有效，等程序运行结束时就会将其销毁。当然，大家也可以将其改为一个具体的数据库名称，比如`“my.db”`，这样就会在项目目录中创建该数据库文件了。下面使用`open()`函数将数据库打开，如果打开失败，则弹出提示对话框。最后使用`QSqlQuery`创建了一个`student`表，并插入了包含`id`和`name`两个属性的五条记录，如下图所示。其中，`id`属性是`int`类型的，`“primary key”`表明该属性是主键，它不能为空，而且不能有重复的值；而`name`属性是`varchar`类型的，并且不大于20个字符。这里使用的SQL语句都要包含在双引号中，如果一行写不完，那么分行后，每一行都要使用两个双引号引起来。

id	name
0	first
1	second
2	third
3	fourth
4	fifth

需要注意，代码中的`query`没有进行任何指定就可以操作前面打开的数据库，这是因为现在只有一个数据库连接，它就是默认连接，这时候所有的操作都是针对该连接的。但是如果要同时操作多个数据库连接，就需要进行指定了，这方面内容可以参考《Qt Creator快速入门》的第17章。

3．下面我们到`main.cpp`中调用连接函数。

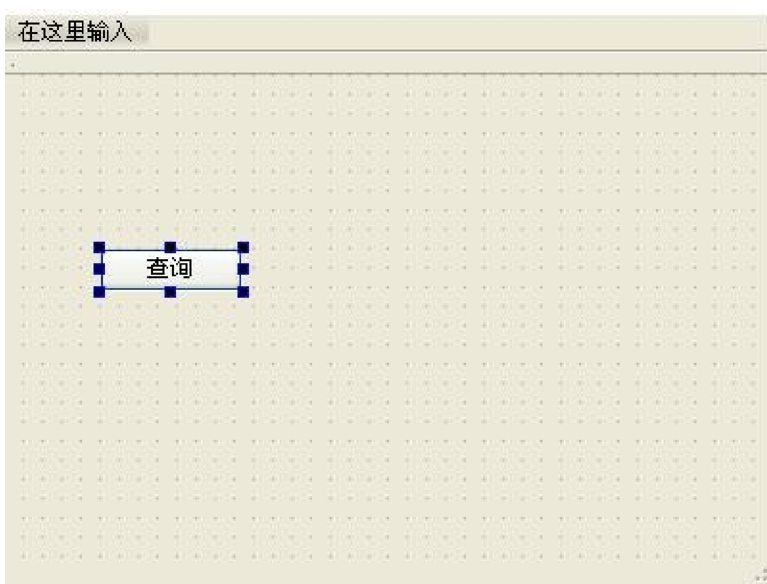
```
#include "mainwindow.h"
#include <QApplication>
#include "connection.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    if (!createConnection())
           return 1;

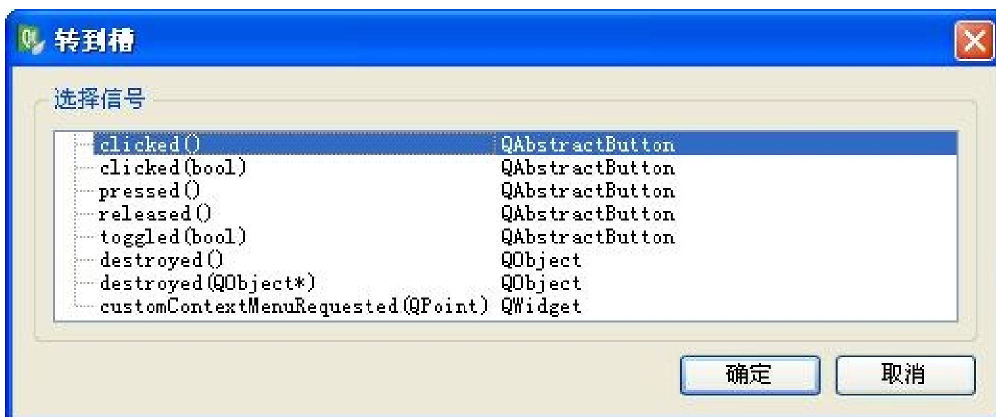
    MainWindow w;
    w.show();

    return a.exec();
}
```

4．我们往界面上添加一个按钮来实现查询操作。双击 `mainwindow.ui` 文件进入设计模式。然后将一个 `Push Button` 拖到界面上，并修改其显示文本为“查询”。效果如下图所示。



5．在查询按钮上点击鼠标右键，选择“转到槽”，然后选择 `clicked()` 单击信号槽并点击确定，如下图所示。



6．将槽的内容更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
       qDebug() << query.value(0).toInt()
                                   << query.value(1).toString();
    }
}
```

7．在 `mainwindow.cpp` 文件中添加头文件：

```
#include <QSqlQuery>
#include <QDebug>
```

8．运行程序，然后按下查询按钮，在应用程序输出窗口将会输出结果，效果如下图所示。



二、操作结果集

在前面的程序中，我们使用 `query.exec("select * from student");` 查询出表中所有的内容。其中的SQL语句 `“select * from student”` 中 `“*”` 号表明查询表中记录的所有属性。而当 `query.exec("select * from student");` 这条语句执行完后，我们便获得了相应的执行结果，因为获得的结果可能不止一条记录，所以称之为结果集。

结果集其实就是查询到的所有记录的集合，在 `QSqlQuery` 类中提供了多个函数来操作这个集合，需要注意这个集合中的记录是从0开始编号的。最常用的操作有：

- `seek(int n)` ：`query`指向结果集的第n条记录；
- `first()` ：`query`指向结果集的第一条记录；
- `last()` ：`query`指向结果集的最后一条记录；
- `next()` ：`query`指向下一条记录，每执行一次该函数，便指向相邻的下一条记录；
- `previous()` ：`query`指向上一条记录，每执行一次该函数，便指向相邻的上一条记录；
- `record()` ：获得现在指向的记录；
- `value(int n)` ：获得属性的值。其中`n`表示你查询的第n个属性，比方上面我们使用`“select * from student`”就相当于`“select id, name from student”`，那么`value(0)`返回`id`属性的值，`value(1)`返回`name`属性的值。该函数返回`QVariant`类型的数据，关于该类型与其他类型的对应关系，可以在帮助中查看QVariant。
- `at()` ：获得现在`query`指向的记录在结果集中的编号。

需要特别注意，刚执行完`query.exec("select *from student");`这行代码时，`query`是指向结果集以外的，我们可以利用`query.next()`使得`query`指向结果集的第一条记录。当然我们也可以利用`seek(0)`函数或者`first()`函数使`query`指向结果集的第一条记录。但是为了节省内存开销，推荐的方法是，在`query.exec("select * from student");`这行代码前加上`query.setForwardOnly(true);`这条代码，此后只能使用`next()`和`seek()`函数。

下面我们通过例子来演示一下这些函数的使用。将槽更改如下：

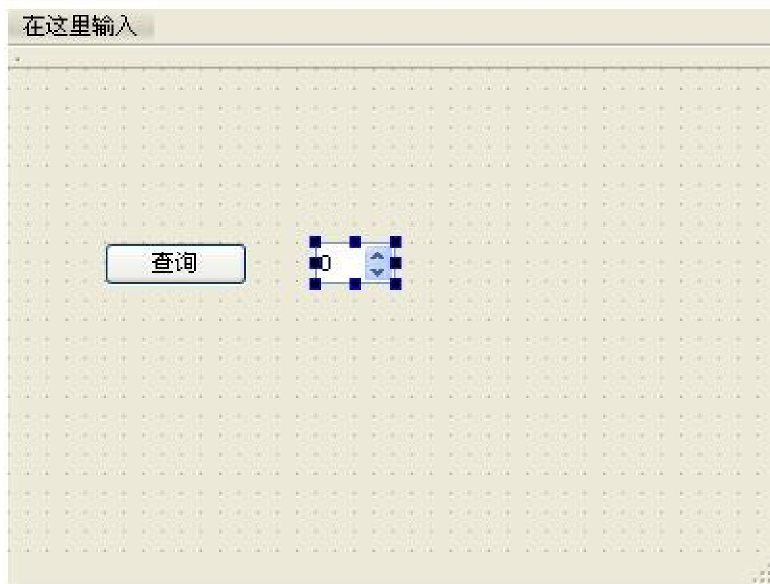
```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery query;
    query.exec("select * from student");
    qDebug() << "exec next() :";
    //开始就先执行一次next()函数，那么query指向结果集的第一条记录
    if(query.next())
    {
       //获取query所指向的记录在结果集中的编号
       int rowNum = query.at();
       //获取每条记录中属性（即列）的个数
       int columnNum = query.record().count();
       //获取"name"属性所在列的编号，列从左向右编号，最左边的编号为0
       int fieldNo = query.record().indexOf("name");
       //获取id属性的值，并转换为int型
       int id = query.value(0).toInt();
       //获取name属性的值
       QString name = query.value(fieldNo).toString();
       //将结果输出
       qDebug() << "rowNum is : " << rowNum
                << " id is : " << id
                << " name is : " << name
                << " columnNum is : " << columnNum;
    }
//定位到结果集中编号为2的记录，即第三条记录，因为第一条记录的编号为0
    qDebug() << "exec seek(2) :";
    if(query.seek(2))
    {
       qDebug() << "rowNum is : " << query.at()
                << " id is : " << query.value(0).toInt()
                << " name is : " << query.value(1).toString();
    }
    //定位到结果集中最后一条记录
    qDebug() << "exec last() :";
    if(query.last())
    {
       qDebug() << "rowNum is : " << query.at()
                << " id is : " << query.value(0).toInt()
                << " name is : " << query.value(1).toString();
    }
}
```

最后在 `mainwindow.cpp` 中添加 `#include <QSqlRecord>` 头文件包含，运行程序，点击查询按钮，输出结果如下图所示。

```
exec next() :
rowNum is :  0  id is :  0  name is :  "first"  columnNum is :  2
exec seek(2) :
rowNum is :  2  id is :  2  name is :  "third"
exec last() :
rowNum is :  4  id is :  4  name is :  "fifth"
```

三、在SQL语句中使用变量

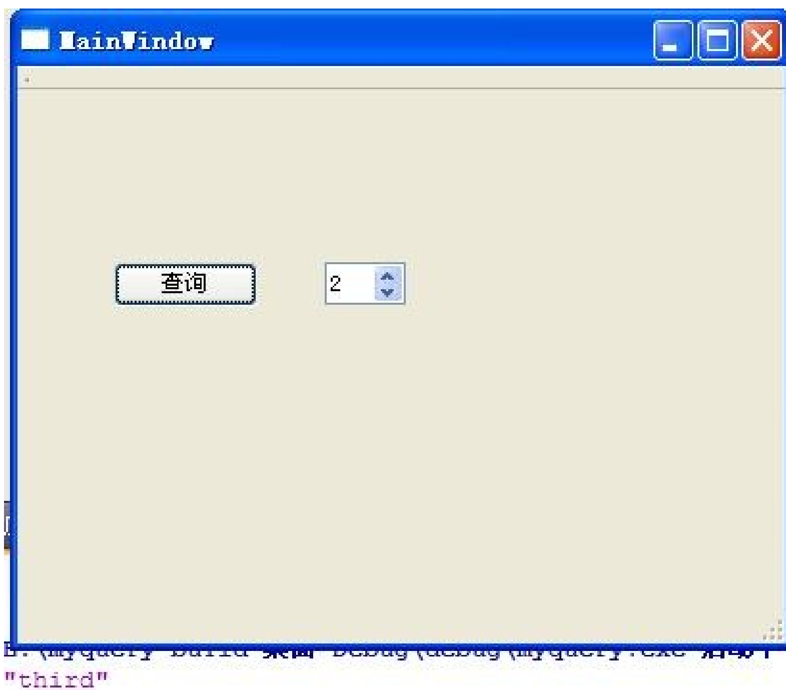
1．我们先来看一个例子。首先在设计模式往界面上添加一个 `Spin Box` 部件，如下图所示。



2 · 将查询按钮槽里面的内容更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery query;
    int id = ui->spinBox->value();
    query.exec(QString("select name from student where id =%1")
                 .arg(id));
    query.next();
    QString name = query.value(0).toString();
    qDebug() << name;
}
```

这里使用了 `QString` 类的 `arg()` 函数实现了在SQL语句中使用变量，我们运行程序，更改 `Spin Box` 的值，然后点击查询按钮，效果如下图所示。



3．其实在 `QSqlQuery` 类中提供了数据绑定同样可以实现在SQL语句中使用变量，虽然它也是通过占位符来实现的，不过使用它形式上更明了一些。下面先来看一个例子，将查询按钮槽更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery query;
    query.prepare("insert into student (id, name) "
                  "values (:id, :name)");
    query.bindValue(0, 5);
    query.bindValue(1, "sixth");
    query.exec();
    query.exec("select * from student");
    query.last();
    int id = query.value(0).toInt();
    QString name = query.value(1).toString();
    qDebug() << id << name;
}
```

这里在 `student` 表的最后又添加了一条记录。然后我们先使用了 `prepare()` 函数，在其中利用了 `“:id”` 和 `“:name”` 来代替具体的数据，而后又利用 `bindValue()` 函数给 `id` 和 `name` 两个属性赋值，这称为绑定操作。其中编号0和1分别代表 `“:id”` 和 `“:name”` ，就是说按照 `prepare()` 函数中出现的属性从左到右编号，最左边是0 。

特别注意，在最后一定要执行 `exec()` 函数，所做的操作才能被真正执行。运行程序，点击查询按钮，可以看到前面添加的记录的信息。这里的 `“:id”` 和 `“:name”` ，叫做占位符，这是ODBC数据库的表示方法，还有一种Oracle的表示方法就是全部用“？”号。例如：

```
query.prepare("insert into student(id, name) "
                  "values (?, ?)");
query.bindValue(0, 5);
query.bindValue(1, "sixth");
query.exec();
```

也可以利用 `addBindValue()` 函数，这样就可以省去编号，它是按顺序给属性赋值的，如下：

```
query.prepare("insert into student(id, name) "
                  "values (?, ?)");
query.addBindValue(5);
query.addBindValue("sixth");
query.exec();
```

当用ODBC的表示方法时，我们也可以将编号用实际的占位符代替，如下：

```
query.prepare("insert into student(id, name) "
                      "values (:id, :name)");
query.bindValue(":id", 5);
query.bindValue(":name", "sixth");
query.exec();
```

以上各种形式的表示方式效果是一样的。

4．下面我们演示一下通过绑定操作在SQL语句中使用变量。更改槽函数如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery query;
    query.prepare("select name from student where id = ?");
    int id = ui->spinBox->value();
    query.addBindValue(id);
    query.exec();
    query.next();
    qDebug() << query.value(0).toString();
}
```

运行程序，可以实现通过 `Spin Box` 的值来进行查询。

四、批处理操作

当要进行多条记录的操作时，我们就可以利用绑定进行批处理。将槽更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQuery q;
    q.prepare("insert into student values (?, ?)");
    QVariantList ints;
    ints << 10 << 11 << 12 << 13;
    q.addBindValue(ints);
    QVariantList names;
    // 最后一个是空字符串，应与前面的格式相同
        names << "xiaoming" << "xiaoliang"
                      << "xiaogang" << QVariant(QVariant::String);
    q.addBindValue(names);
    if (!q.execBatch()) //进行批处理，如果出错就输出错误
       qDebug() << q.lastError();
    //下面输出整张表
    QSqlQuery query;
    query.exec("select * from student");
    while(query.next())
    {
       int id = query.value(0).toInt();
       QString name = query.value(1).toString();
       qDebug() << id << name;
    }
}
```

然后需要在 `mainwindow.cpp` 上添加头文件包含： `#include <QSqlError>` 。我们在程序中利用列表存储了同一属性的多个值，然后进行了值绑定。最后执行 `execBatch()` 函数进行批处理。注意程序中利用 `QVariant(QVariant::String)` 来输入空值 `NULL` ，因为前面都是 `QString` 类型的，所以这里要使用 `QVariant::String` 使格式一致化。 运行程序，效果如下图所示：

```
0 "first"
1 "second"
2 "third"
3 "fourth"
4 "fifth"
10 "xiaoming"
11 "xiaoliang"
12 "xiaogang"
13 ""
```

五、事务操作

事务可以保证一个复杂的操作的原子性，就是对于一个数据库操作序列，这些操作要么全部做完，要么一条也不做，它是一个不可分割的工作单位。在Qt中，如果底层的数据库引擎支持事务，那么 `QSqlDriver::hasFeature(QSqlDriver::Transactions)` 会返回 `true` 。可以使用 `QSqlDatabase::transaction()` 来启动一个事务，然后编写一些希望在事务中执行的SQL语句，最后调用 `QSqlDatabase::commit()` 或者 `QSqlDatabase::rollback()` 。当使用事务时必须在创建查询以前就开始事务，例如：

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROMemployee WHERE name = 'Torild Halvorsen'");
if (query.next()) {
    int employeeId = query.value(0).toInt();
    query.exec("INSERT INTO project(id, name, ownerid) "
               "VALUES (201, 'ManhattanProject', "
               + QString::number(employeeId) + ')');
}
QSqlDatabase::database().commit();
```

结语

对执行SQL语句我们就介绍这么多，其实Qt中提供了更为简单的不需要SQL语句就可以操作数据库的方法，我们在下一节讲述这些内容。

涉及到的源码:

- myquery01.zip
- myquery02.zip

第24篇 数据库（四）SQL查询模型`QSqlQueryModel`

导语

在上一篇的最后我们讲到，Qt中使用了自己的机制来避免使用SQL语句，为我们提供了更简单的数据库操作及数据显示模型，分别是只读的`QSqlQueryModel`，操作单表的`QSqlTableModel`和以及可以支持外键的`QSqlRelationalTableModel`。这次我们先讲解`QSqlQueryModel`。

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

正文

一、简单的查询操作

1．新建Qt Gui应用，项目名称为`queryModel`，基类为`QMainWindow`，类名为`MainWindow`。

2．完成后打开`queryModel.pro`，将第一行代码更改为：

```
QT       += core gui sql
```

然后保存该文件。

3．往项目中添加新的C++头文件，名称为`connection.h`，完成后将其内容更改如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open()) return false;
    QSqlQuery query;
    query.exec("create table student (id int primary key, name vchar)");
    query.exec("insert into student values (0,'yafei0')");
    query.exec("insert into student values (1,'yafei1')");
    query.exec("insert into student values (2,'yafei2')");
    return true;
}
#endif // CONNECTION_H
```

这里使用了 `db.setDatabaseName("database.db");` ，我们没有再使用以前的内存数据库，而是使用了真实的文件，这样后面对数据库进行的操作就能保存下来了。

4. 然后进入 `main.cpp` 文件，更改如下：

```
#include "mainwindow.h"
#include <QApplication>
#include "connection.h"

int main(int argc, char *argv[])
{
     QApplication a(argc, argv);
     if(!createConnection())
          return 1;
     MainWindow w;
     w.show();
     return a.exec();
}
```

5. 下面进入设计模式，向界面上拖入一个 `Push Button` 按钮，更改显示文本为“查询”，然后进入其单击信号槽，更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("select * from student");
    model->setHeaderData(0, Qt::Horizontal, tr("id"));
    model->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view = new QTableView;
    view->setModel(model);
    view->show();
}
```

这里新建了 `QSqlQueryMode` l类对象 `model` ，并用 `setQuery()` 函数执行了SQL语句 `“("select * fromstudent");”` 用来查询整个 `student` 表的内容，可以看到，该类并没有完全避免SQL语句。然后我们设置了表中属性显示时的名字。最后我们建立了一个视图 `view` ，并将这个 `model` 模型关联到视图中，这样数据库中的数据就能在窗口上的表中显示出来了。

6. 在 `mainwindow.cpp` 中添加头文件包含：

```
#include <QSqlQueryModel>
#include <QTableView>
```

7．运行程序，按下查询按钮，效果如下图所示。



8．我们查看一下编译生成的目录（我这里是 `E:\queryModel-build-桌面-Debug` ），这里面有生成的数据库文件，如下图所示。



二、`QSqlQueryModel` 常用操作

1．在查询按钮槽中继续添加如下代码：

```
int column= model->columnCount(); //获得列数
int row = model->rowCount();    // 获得行数
QSqlRecord record = model->record(1); //获得一条记录
QModelIndex index = model->index(1,1);   //获得一条记录的一个属性的值
qDebug() << "column numis:" << column << endl
        << "row num is:" << row << endl
        <<"the second record is:" << record << endl
        << "the data of index(1,1) is:"<< index.data();
```

2、然后在 `mainwindow.cpp` 文件中添加下面的头文件包含：

```
#include <QSqlRecord>
#include <QModelIndex>
#include <QDebug>
```

3、运行程序，点击查询按钮，输出内容如下图所示。

```
column num is: 2
row num is: 3
the second record is: QSqlRecord( 2 )
 " 0:" QSqlField("id", int, generated: yes, typeID: 1) "1"
 " 1:" QSqlField("name", QString, generated: yes, typeID: 3) "yafei1"
the data of index(1,1) is: QVariant(QString, "yafei1")
```

4、另外我们可以直接使用上一节讲到的 `QSqlQuery` 来执行SQL语句，例如：

```
QSqlQuery query = model->query();
query.exec("select name from studentwhere id = 2 ");
query.next();
qDebug() << query.value(0).toString();
```

5、我们将查询按钮槽更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("select * from student");
    model->setHeaderData(0, Qt::Horizontal, tr("id"));
    model->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view = new QTableView;
    view->setModel(model);
    view->show();

    QSqlQuery query = model->query();
    query.exec("insertinto student values (10,'yafei10')");
}
```

这里使用 `query` 向表中添加了一条记录。

6、在 `mainwindow.cpp` 中添加头文件 `#include <QSqlQuery>` ，然后运行程序，发现最后添加的记录并没有显示出来，当关闭程序，再次运行的时候才显示出来，效果如下图所示。



7．为什么会出现上面的情况呢？那是因为前面我们执行了添加记录的SQL语句，但是在添加记录之前，查询结果就已经显示了，所以我们的更新没能动态的显示出来。为了能让其动态地显示我们的更新，可以将槽最后的代码更改如下：

```
QSqlQuery query = model->query();
query.exec("insert into student values (20,'yafei20')");
model->setQuery("select * from student"); //再次查询整张表
view->show(); //再次进行显示
```

这里我们修改完表以后，再次进行了查询并显示。大家可以运行程序，发现新的记录可以直接显示出来了。

三、创建自定义 `QSqlQueryModel`

前面我们讲到这个模型默认是只读的，所以在窗口上并不能对表格中的内容进行修改。但是我们可以创建自己的模型，然后按照自己的意愿来显示数据和修改数据。要想使其可读写，需要自己的类继承自 `QSqlQueryModel` ，并且重写 `setData()` 和 `flags()` 两个函数。如果我们要改变数据的显示，就要重写 `data()` 函数。

下面的例子中我们让 `student` 表查询结果的 `id` 属性列显示红色， `name` 属性列可编辑。

1．向项目中添加新的C++类，类名为 `MySqlQueryModel` ，基类为 `QSqlQueryModel` ，类型信息选择“继承自 `QObject` ”。

2．完成后打开 `mysqlquerymodel.h` 文件，在 `public` 中添加函数声明：

```
Qt::ItemFlags flags(const QModelIndex &index) const;
bool setData(const QModelIndex &index, const QVariant &value, int role);
QVariant data(const QModelIndex &item, int role=Qt::DisplayRole) const;
```

然后添加私有函数声明：

```
private:
    bool setName(int studentId, const QString &name);
    void refresh();
```

3・到 `mysqlquerymodel.cpp` 文件中，更改如下：

```
#include "mysqlquerymodel.h"
#include <QSqlQuery>
#include <QColor>
MySqlQueryModel::MySqlQueryModel(QObject *parent) :
    QSqlQueryModel(parent)
{
}

Qt::ItemFlags MySqlQueryModel::flags(
        const QModelIndex &index) const //返回表格是否可更改的标志
{
    Qt::ItemFlags flags = QSqlQueryModel::flags(index);
    if (index.column() == 1) //第二个属性可更改
       flags |= Qt::ItemIsEditable;
    return flags;
}

bool MySqlQueryModel::setData(const QModelIndex &index, const QVariant &value, int /*
role */)
        //添加数据
{
    if (index.column() < 1 || index.column() > 2)
       return false;
   QModelIndex primaryKeyIndex = QSqlQueryModel::index(index.row(), 0);
    int id = data(primaryKeyIndex).toInt(); //获取id号
    clear();
    bool ok;
    if (index.column() == 1) //第二个属性可更改
       ok = setName(id, value.toString());
    refresh();
    return ok;
}

void MySqlQueryModel::refresh() //更新显示
{
    setQuery("select * from student");
    setHeaderData(0, Qt::Horizontal, QObject::tr("id"));
    setHeaderData(1, Qt::Horizontal, QObject::tr("name"));
}

//添加name属性的值
bool MySqlQueryModel::setName(int studentId, const QString &name)
{
    QSqlQuery query;
    query.prepare("update student set name = ? where id = ?");
    query.addBindValue(name);
    query.addBindValue(studentId);
    return query.exec();
}

//更改数据显示样式
QVariant MySqlQueryModel::data(const QModelIndex &index, int role) const
{
QVariant value = QSqlQueryModel::data(index, role);

//第一个属性的字体颜色为红色
    if (role == Qt::TextColorRole && index.column() == 0)
       return qVariantFromValue(QColor(Qt::red));
return value;
}
```

```
4・到mainwindow.cpp文件中先添加头文件包含：
#include "mysqlquerymodel.h"


5・更改查询按钮槽内容如下：
void MainWindow::on_pushButton_clicked()
{
    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("select * from student");
    model->setHeaderData(0, Qt::Horizontal, tr("id"));
    model->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view = new QTableView;
    view->setModel(model);
    view->show();

//创建自己模型的对象
    MySqlQueryModel *myModel = new MySqlQueryModel;    myModel->setQuery("select * fro
m student");
    myModel->setHeaderData(0, Qt::Horizontal, tr("id"));
    myModel->setHeaderData(1, Qt::Horizontal, tr("name"));
    QTableView *view1 = new QTableView;
    view1->setWindowTitle("mySqlQueryModel"); //修改窗口标题
    view1->setModel(myModel);
    view1->show();
}
```

运行程序，效果如下图所示。



结语

本节讲解了 `QSqlQueryModel` 的相关内容，该类默认是一个只读的SQL语句查询模型，不过可以对其进行重写来实现编辑功能。下一节我们将讲解封装更好的 `QSqlTableModel` 模型，它已经基本上摆脱了SQL语句。

涉及到的源码

第25篇 数据库（五）SQL表格模型 `QSqlTableModel`

导语

在上一篇我们讲到只读的 `QsqlQueryModel` 模型其实也可以实现编辑功能的，但是实现起来很麻烦。而 `QSqlTableModel` 提供了一个一次只能操作单个SQL表的读写模型，它是 `QSqlQuery` 的更高层次的替代品，可以浏览和修改独立的SQL表，并且只需编写很少的代码，而且不需要了解SQL语法。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

一、创建数据库

1．新建Qt Gui应用，项目名称为 `tableModel` ，基类 `QMainWindow` ，类名 `MainWindow` 。

2．完成后打开 `tableModel.pro` 文件，将第一行代码更改为：

```
QT       += coregui sql
```

然后保存文件。

3．向项目中添加新的C++头文件，名称为 `connection.h` 。完成后将其内容更改如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open()) return false;
    QSqlQuery query;
    query.exec(QString(
      "create table student (id int primary key, name vchar)"));
    query.exec(QString("insert into student values (0,'刘明')"));
    query.exec(QString("insert into student values (1,'陈刚')"));
    query.exec(QString("insert into student values (2,'王红')"));
    return true;
}
#endif // CONNECTION_H
```

这里因为语句中使用了中文，所以使用了 `QString()` 进行编码转换，这个还需要在 `main()` 函数中设置编码。

4．下面将 `main.cpp` 文件更改如下：

```
#include "mainwindow.h"
#include <QApplication>
#include "connection.h"
#include <QTextCodec>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("utf8"));
    QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
    if(!createConnection())
       return 1;
    MainWindow w;
    w.show();

    return a.exec();
}
```

这里的 `setCodecForCStrings()` 就是用来设置字符串编码的。

5．下面进入设计模式，向窗口上拖入 `Label` 、 `Push Button` 、 `Line Edit` 和 `Table View` 等部件，进行界面设计，效果如下图所示。



6．完成后到 `mainwindow.h` 文件中，先包含头文件：

```
#include <QSqlTableModel>
```

然后添加私有对象声明：

```
QSqlTableModel *model;
```

7．到 `mainwindow.cpp` ，在构造函数添加如下代码：

```
model = new QSqlTableModel(this);
model->setTable("student");
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
model->select(); //选取整个表的所有行
//不显示name属性列,如果这时添加记录，则该属性的值添加不上
// model->removeColumn(1);
ui->tableView->setModel(model);
//使其不可编辑
//ui->tableView->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

这里创建一个 `QSqlTableModel` 后，只需使用 `setTable()` 来为其指定数据库表，然后使用 `select()` 函数进行查询，调用这两个函数就等价于执行了 `“select * from student”` 这个SQL语句。这里还可以使用 `setFilter()` 来指定查询时的条件，在后面会看到这个函数的使用。在使用该模型以前，一般还要设置其编辑策略，它由 `QSqlTableModel::EditStrategy` 枚举变量定义，一共有三个值，如下图所示。用来说明当数据库中的值被编辑后，什么情况下提交修改。

常量	描述
QSqlTableModel::OnFieldChange	所有对模型的改变都会立即应用到数据库
QSqlTableModel::OnRowChange	对一条记录的改变会在用户选择另一条记录时被应用
QSqlTableModel::OnManualSubmit	所有的改变都会在模型中进行缓存，直到调用 submitAll() 或者 revertAll()函数

运行程序，效果如下图所示。



可以看到，这个模型已经完全脱离了SQL语句，我们只需要执行 `select()` 函数就能查询整张表。上面有两行代码被注释掉了，你可以取消注释，测试一下它们的作用。

二、修改操作

1．我们进入"提交修改"按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_3_clicked()
{
    model->database().transaction(); //开始事务操作
    if (model->submitAll()) {
       model->database().commit(); //提交
    } else {
       model->database().rollback(); //回滚
       QMessageBox::warning(this, tr("tableModel"),
                            tr("数据库错误: %1")
                            .arg(model->lastError().text()));
    }
}
```

这里用到了事务操作，真正起提交操作的是 `model->submitAll()` 一句，它提交所有更改。

2．进入"撤销修改"按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_4_clicked()
{
    model->revertAll();
}
```

3．在 `mainwindow.cpp` 文件中包含头文件：

```
#include <QMessageBox>
#include <QSqlError>
```

4．现在运行程序，我们将“陈刚”改为“李强”，如果我们点击“撤销修改”，那么它就会重新改为“陈刚”，而当我们点击“提交修改”后它就会保存到数据库，此时再点击“撤销修改”就修改不回来了。

可以看到，这个模型可以将所有修改先保存到 `model` 中，只有当我们执行提交修改后，才会真正写入数据库。当然这也是因为我们在最开始设置了它的保存策略：

```
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
```

这里的 `OnManualSubmit` 表明我们要提交修改才能使其生效。

三、查询操作

1．进入“查询”按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_clicked()
{
    QString name = ui->lineEdit->text();
    //根据姓名进行筛选
    model->setFilter(QString("name = '%1'").arg(name));
    //显示结果
    model->select();
}
```

使用 `setFilter()` 函数进行关键字筛选，这个函数是对整个结果集进行查询。

2．进入“显示全表”按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_2_clicked()
{
    model->setTable("student");   //重新关联表
    model->select();   //这样才能再次显示整个表的内容
}
```

为了再次显示整个表的内容，我们需要再次关联这个表。

3．下面运行程序，输入一个姓名，点击“查询”按钮后，就可以显示该记录了。再点击“显示全表”按钮则返回。如下图所示。



四、排序操作

分别进入“按id升序排序”和“按id降序排序”按钮的单击信号槽，更改如下：

```
// 升序
void MainWindow::on_pushButton_7_clicked()
{
    model->setSort(0, Qt::AscendingOrder); //id属性即第0列，升序排列
    model->select();
}
// 降序
void MainWindow::on_pushButton_8_clicked()
{
    model->setSort(0, Qt::DescendingOrder);
    model->select();
}
```

这里使用了 `setSort()` 函数进行排序，它有两个参数，第一个参数表示按第几个属性排序，表头从左向右，最左边是第0个属性，这里就是id属性。第二个参数是排序方法，有升序和降序两种。运行程序，效果如下图所示。



五、删除操作

我们进入“删除选中行”按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_6_clicked()
{
    //获取选中的行
    int curRow = ui->tableView->currentIndex().row();

    //删除该行
    model->removeRow(curRow);

    int ok = QMessageBox::warning(this,tr("删除当前行!"),tr("你确定"
                                                         "删除当前行吗？"),
                                  QMessageBox::Yes,QMessageBox::No);
    if(ok == QMessageBox::No)
    {
       model->revertAll(); //如果不删除，则撤销
    }
    else model->submitAll(); //否则提交，在数据库中删除该行
}
```

删除行的操作会先保存在 `model` 中，当我们执行了 `submitAll()` 函数后才会真正的在数据库中删除该行。这里我们使用了一个警告框来让用户选择是否真得要删除该行。运行程序，效果如下图所示。



我们点击第二行，然后单击“删除选中行”按钮，出现了警告框。这时你会发现，表中的第二行前面出现了一个小感叹号，表明该行已经被修改了，但是还没有真正的在数据库中修改，这时的数据有个学名叫脏数据(Dirty Data)。当我们按钮“Yes”按钮后数据库中的数据就会被删除，如果按下“No”，那么更改就会取消。

六、插入操作

我们进入“添加记录”按钮的单击信号槽，更改如下：

```
void MainWindow::on_pushButton_5_clicked()
{
    int rowNum = model->rowCount(); //获得表的行数
    int id = 10;
    model->insertRow(rowNum); //添加一行
    model->setData(model->index(rowNum,0),id);
    //model->submitAll(); //可以直接提交
}
```

在表的最后添加一行，因为在 `student` 表中我们设置了 `id` 号是主键，所以这里必须使用 `setData()` 函数给新加的行添加 `id` 属性的值，不然添加行就不会成功。这里可以直接调用 `submitAll()` 函数进行提交，也可以利用“提交修改”按钮进行提交。运行程序，效果如下图所示。



按下“添加记录”按钮后，就添加了一行，不过在该行的前面有个星号，如果我们按下“提交修改”按钮，这个星号就会消失。当然，如果我们将上面代码里的提交函数的注释去掉，也就不会有这个星号了。

结语

可以看到这个模型很强大，而且完全脱离了SQL语句，就算你不怎么懂数据库知识，也可以利用它进行大部分常用的操作。我们也看到了，这个模型提供了缓冲区，可以先将修改保存起来，当我们执行提交函数时，再去真正地修改数据库。当然，这个模型比前面的模型更高级，前面讲的所有操作，在这里都能执行。

涉及到的源码

第26篇 数据库（六）SQL关系表格模型 `QSqlRelationalTableModel`

导语

`QSqlRelationalTableModel` 继承自 `QSqlTableModel` ，并且对其进行了扩展，提供了对外键的支持。一个外键就是一个表中的一个属性和其他表中的主键属性之间的一对一的映射。例如， `student` 表中的 `course` 属性对应的是 `course` 表中的 `id` 属性，那么就称属性 `course` 是一个外键。因为这里的 `course` 属性的值是一些数字，这样的显示很不友好，使用关系表格模型，就可以将它显示为 `course` 表中的 `name` 属性的值。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

一、使用外键

1．新建Qt Gui应用，名称为 `relationalTableModel` ，基类为 `QMainWindow` ，类名为 `MainWindow` 。完成后打开 `relationalTableModel.pro` 项目文件，将第一行改为：

```
QT       += coregui sql
```

然后保存该文件。

2．下面向项目中添加新的C++头文件 `connection.h` ，并更改其内容如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include <QSqlDatabase>
#include <QSqlQuery>
static bool createConnection()
{
   QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
   db.setDatabaseName("database.db");
   if(!db.open()) return false;
   QSqlQuery query;
   query.exec("create table student (id int primary key, name vchar,course int)");
   query.exec("insert into student values(1,'yafei0',1)");
   query.exec("insert into student values(2,'yafei1',1)");
   query.exec("insert into student values(3,'yafei2',2)");

query.exec("create table course (id int primarykey, name vchar, teacher vchar)");
   query.exec("insert into course values(1,'Math','yafeilinux1')");
   query.exec("insert into course values(2,'English','yafeilinux2')");
   query.exec("insert into course values(3,'Computer','yafeilinux3')");
   return true;
}
#endif // CONNECTION_H
```

在这里建立了两个表， `student` 表中有一项是 `course` ，它是 `int` 型的，而 `course` 表的主键也是int型的。如果要将 `course` 项和 `course` 表进行关联，它们的类型就必须相同，一定要注意这一点。

3．更改 `main.cpp` 文件内容如下：

```
#include "mainwindow.h"
#include <QApplication>
#include "connection.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if(!createConnection()) return 1;
    MainWindow w;
    w.show();

    return a.exec();
}
```

4．然后到 `mainwindow.h` 文件中，先包含头文件：

```
#include<QSqlRelationalTableModel>
```

然后添加 `private` 类型对象声明：

```
QSqlRelationalTableModel *model;
```

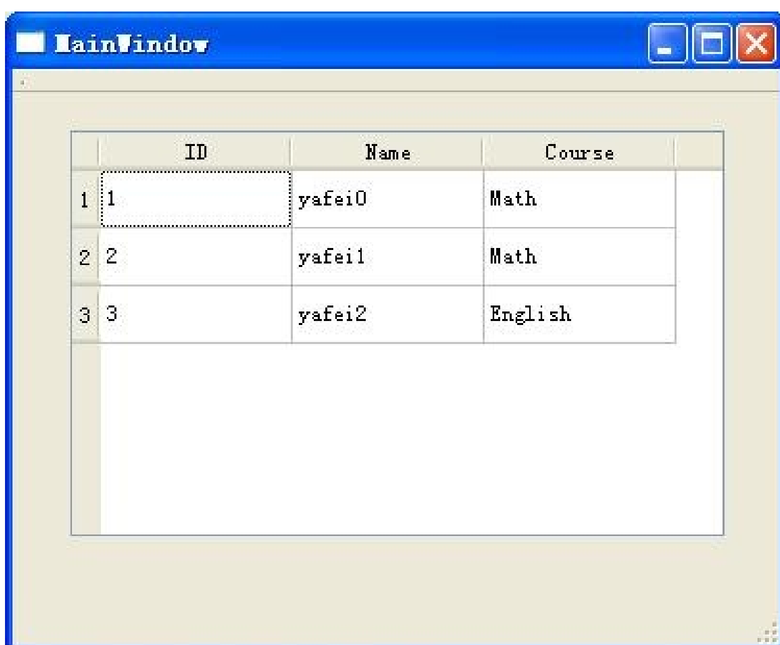
5．到设计模式，往界面上拖放一个 `Table View` 部件。

6．到 `mainwindow.cpp` 文件中，在构造函数里添加如下代码：

```
model = new QSqlRelationalTableModel(this);
//属性变化时写入数据库
model->setEditStrategy(QSqlTableModel::OnFieldChange);
model->setTable("student");
//将student表的第三个属性设为course表的id属性的外键，
//并将其显示为course表的name属性的值
model->setRelation(2,QSqlRelation("course","id","name"));
model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
model->setHeaderData(1, Qt::Horizontal, QObject::tr("Name"));
model->setHeaderData(2, Qt::Horizontal, QObject::tr("Course"));
model->select();
ui->tableView->setModel(model);
```

这里修改了 `model` 的提交策略， `OnFieldChange` 表示只要属性被改动就马上写入数据库，这样就不需要我们再执行提交函数了。 `setRelation()` 函数实现了创建外键，注意它的格式就行了。

7．运行程序，效果如下图所示。



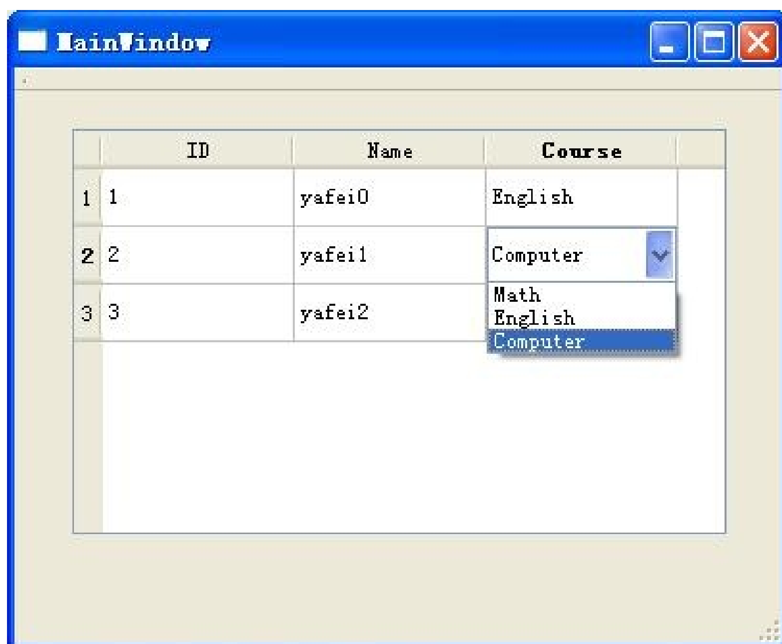
可以看到 `Course` 属性已经不再是编号，而是具体的课程了。关于外键，大家也应该有一定的认识了吧，说简单点就是将两个相关的表建立一个桥梁，让它们关联起来。

二、使用委托

有时我们也希望，如果用户更改课程属性，那么只能在课程表中有的课程中进行选择，而不能随意填写课程。Qt中还提供了一个 `QSqlRelationalDelegate` 委托类，它可以为 `QSqlRelationalTableModel` 显示和编辑数据。这个委托为一个外键提供了一个 `QComboBox` 部件来显示所有可选的数据，这样就显得更加人性化了。使用这个委托是很简单的，我们先在 `mainwindow.cpp` 文件中添加头文件 `#include <QSqlRelationalDelegate>` ，然后继续在构造函数中添加如下一行代码：

```
ui->tableView->setItemDelegate(
                new QSqlRelationalDelegate(ui->tableView));
```

运行程序，效果如下图所示。



结语

我们可以根据自己的需要来选择使用哪个模型。如果熟悉SQL语法，又不需要将所有的数据都显示出来，那么只需要使用 `QSqlQuery` 就可以了。对于 `QSqlTableModel` ，它主要是用来显示一个单独的表格的，而 `QSqlQueryModel` 可以用来显示任意一个结果集，如果想显示任意一个结果集，而且想使其可读写，那么建议子类化 `QSqlQueryModel` ，然后重新实现 `flags()` 和 `setData()` 函数。更多相关内容请查看《Qt Creator快速入门》第17章。

涉及到的源码下载

第27篇 XML（一）使用DOM读取XML文档

导语 XML(ExtensibleMarkup Language，可扩展标记语言)，是一种类似于HTML的标记语言，但它的设计目的是用来传输数据，而不是显示数据。XML的标签没有被预定义，用户需要在使用时自行进行定义。XML是W3C（万维网联盟）的推荐标准。相对于数据库表格的二维表示，XML使用的树形结构更能表现出数据的包含关系，作为一种文本文件格式，XML简单明了的特性使得它在信息存储和描述领域非常流行。

在Qt中提供了QtXml模块来进行XML文档的处理，我们在Qt帮助中输入关键字QtXml Module，可以看到该模块的类表。这里主要提供了三种解析方法： DOM方法，可以进行读写；SAX方法，可以进行读取；基于流的方法，分别使用 `QXmlStreamReader` 和 `QXmlStreamWriter` 进行读取和写入。要在项目中使用 `QtXml` 模块，还需要在项目文件（ `.pro` 文件）中添加 `QT += xml` 一行代码。这一节我们先来讲解一下DOM的方法。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

正文

一、XML文档示例

下面是一个规范的XML文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
   <book id="01">
       <title>Qt</title>
       <author>shiming</author>
    </book>
   <book id="02">
       <title>Linux</title>
       <author>yafei</author>
   </book>
</library>
```

每个XML文档都由XML说明（或者称为XML序言）开始，它是对XML文档处理的环境和要求的说明，比如这里的 `<?xmlversion="1.0" encoding="UTF-8"?>` ，其中 `xml version=“1.0”` ，表明使用的XML版本号，这里字母是区分大小写的； `encoding=“UTF-8”` 是使用的编码，指出文档是使用何种字符集建立的，默认值为Unicode编码。XML文档内容由多个元素组成，一个元素由起始标签 `<标签名>` 和终止标签 `</标签名>` 以及两个标签之间的内容组成，而文档中第一个

元素被称为根元素，比如这里的`<library></library>`，XML文档必须有且只有一个根元素。元素的名称是区分大小写的，元素还可以嵌套，比如这里的`library`、`book`、`title`和`author`等都是元素。元素可以包含属性，用来描述元素的相关信息，属性名和属性值在元素的起始标签中给出，格式为`<元素名 属性名="属性值">`，如`<book id="01">`，属性值必须在单引号或者双引号中。在元素中可以包含子元素，也可以只包含文本内容，比如这里的`<title>Qt</title>`中的Qt就是文本内容。

二、使用DOM读取XML文档内容

Dom（Document Object Model，即文档对象模型）把XML文档转换成应用程序可以遍历的树形结构，这样便可以随机访问其中的节点。它的缺点是需要将整个XML文档读入内存，消耗内存较多。

在Qt中使用`QDomProcessingInstruction`类来表示XML说明，元素对应`QDomElement`类，属性对应`QDomAttr`类，文本内容由`QDomText`类表示。所有的DOM节点，比如这里的说明、元素、属性和文本等，都使用`QDomNode`来表示，然后使用对应的`isProcessingInstruction()`、`isElement()`、`isAttr()`和`isText()`等函数来判断是否是该类型的元素，如果是，那么就可以使用`toProcessingInstruction()`、`toElement()`、`toAttr()`和`toText()`等函数转换为具体的节点类型。

下面来演示一个例子，将读取前面介绍的XML文档的内容。

1 · 新建Qt控制台应用，项目名称为`myDom`。

2 · 完成后打开`myDom.pro`项目文件，将第一行代码更改为：

```
QT       += core    xml
```

然后保存该文件。

3 · 打开`main.cpp`文件，更改内容如下：

```
#include <QCoreApplication>
#include <QtXml>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    // 新建QDomDocument类对象，它代表一个XML文档
    QDomDocument doc;
    // 建立指向“my.xml”文件的QFile对象
    QFile file("my.xml");
     // 以只读方式打开
     if (!file.open(QIODevice::ReadOnly)) return 0;
     // 将文件内容读到doc中
     if (!doc.setContent(&file))
    { file.close(); return 0; }
     // 关闭文件
    file.close();
    // 获得doc的第一个节点，即XML说明
     QDomNode firstNode = doc.firstChild();
     // 输出XML说明
     qDebug() << firstNode.nodeName()    << firstNode.nodeValue();

     return a.exec();
}
```

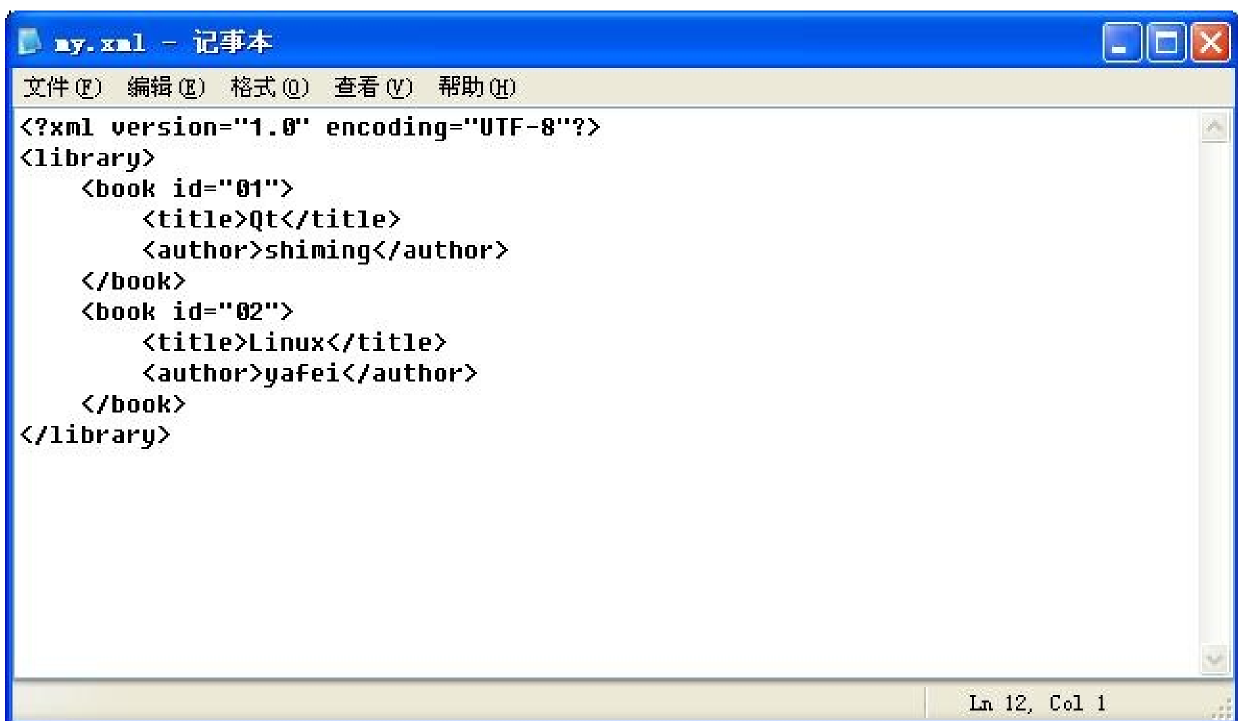
4．然后先点击一下Qt Creator左下角的锤子图标来构建项目，这样会在源码目录旁生成构建目录，比如这里是 `myDom-build-桌面-Debug` ，我们进入该目录，然后新建一个文本文档，如下图所示。



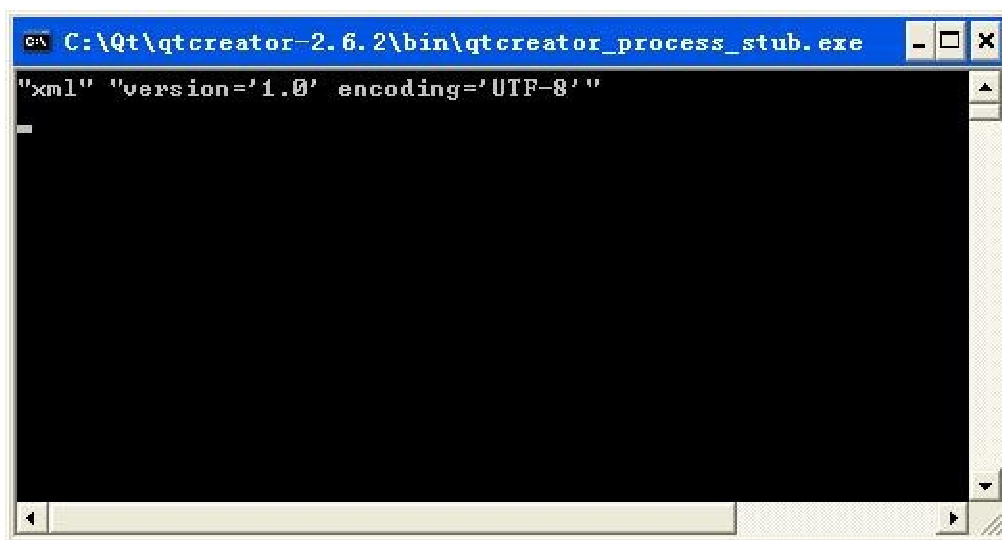
下面将其文件名更改为my.xml，注意一定要更改扩展名，一些电脑中扩展名可能自动隐藏了，可以去“工具→文件夹选项→查看”中修改。提示信息选择“是”即可。如下图所示。



更改完成后使用记事本打开 `my.xml` 文件，然后将前面的xml文档内容添加进去，保存退出即可。如下图所示。



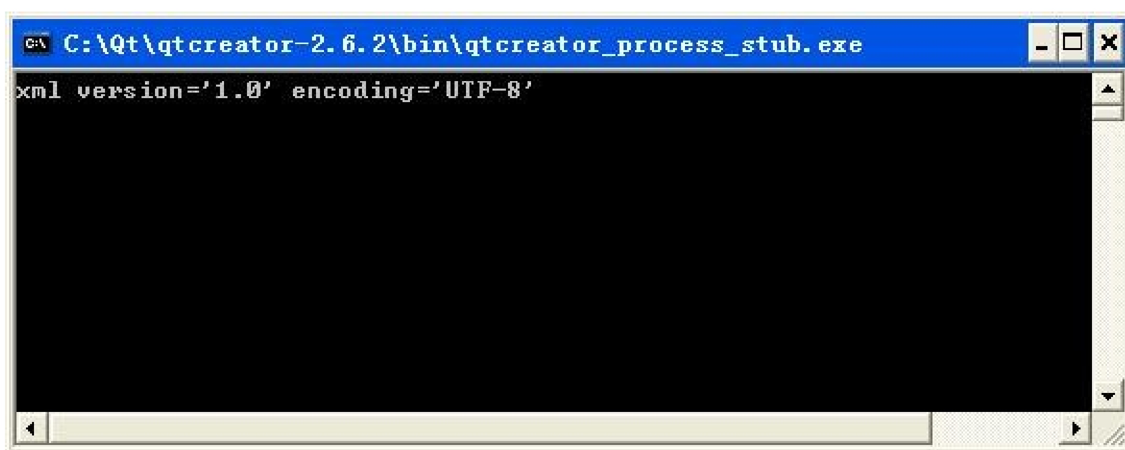
5．现在运行程序，效果如下图所示。



如果大家不愿意看到字符串两边的引号，可以将源码中得 `qDebug()` 语句更改如下：

```
qDebug() << qPrintable(firstNode.nodeName())
             << qPrintable(firstNode.nodeValue());
```

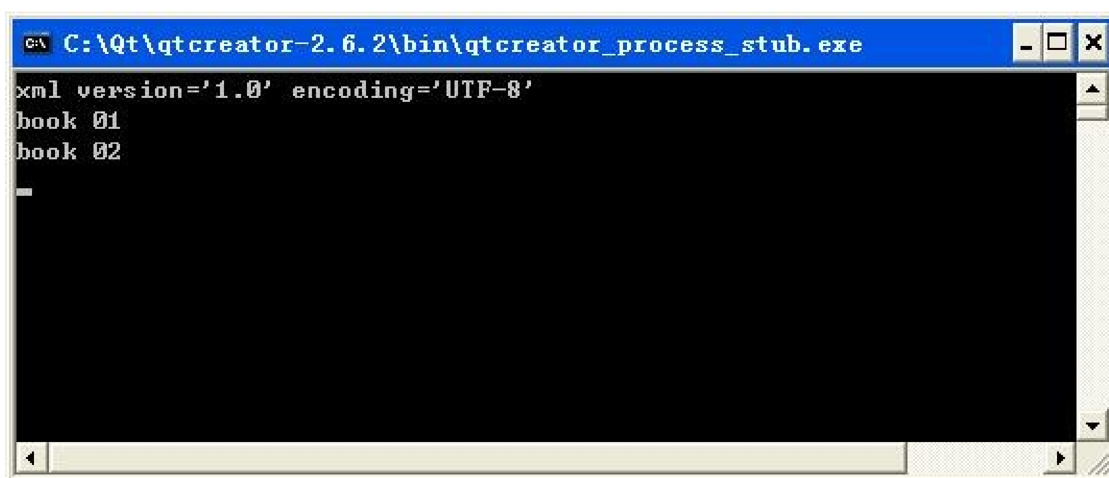
运行程序，效果如下图所示。



6．下面在 `main()` 函数的 `return a.exec();` 一行代码前继续添加如下代码：

```
QDomElement docElem = doc.documentElement();  //返回根元素
QDomNode n = docElem.firstChild();   //返回根节点的第一个子节点
//如果节点不为空
while(!n.isNull())
{
   if (n.isElement())  //如果节点是元素
   {
      QDomElement e = n.toElement();  //将其转换为元素
      qDebug() << qPrintable(e.tagName())   //返回元素标记
         << qPrintable(e.attribute("id"));  //返回元素id属性的值
    }
      n = n.nextSibling();  //下一个兄弟节点
}
```

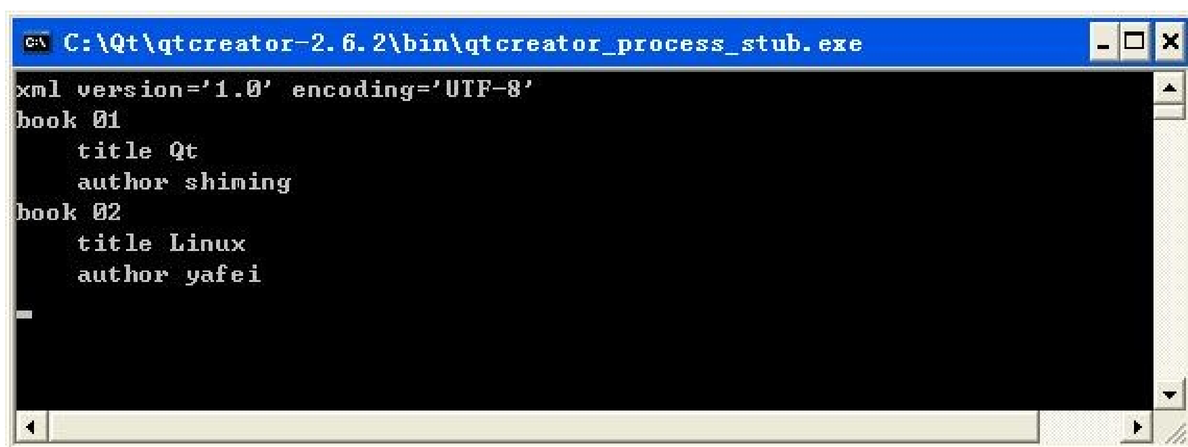
这里使用了 `firstChild()` 函数和 `nextSibling()` 函数，然后利用 `while()` 循环来实现对所有子元素的遍历。运行程序，效果如下图所示。



7．下面更改源码中得 `if()` 语句的内容，输出所有子节点的内容：

```
if (n.isElement())  //如果节点是元素
{
    QDomElement e = n.toElement();
    qDebug() << qPrintable(e.tagName())
            << qPrintable(e.attribute("id"));
    // 获得元素e的所有子节点的列表
    QDomNodeList list = e.childNodes();
    // 遍历该列表
    for(int i=0; i<list.count(); i++)
    {
       QDomNode node = list.at(i);
       if(node.isElement())
           qDebug() << "    " << qPrintable(node.toElement().tagName())
                              <<qPrintable(node.toElement().text());
    }
}
```

这里使用了 `childNodes()` 函数获得了元素所有子节点的列表，然后通过遍历这个列表实现了遍历其所有子元素。运行程序，效果如下图所示。



结语

通过上面的例子，我们实现了对一个XML文档的读取。可以看到，在 `QDom` 中，是将整个XML文件读到内存中的 `doc` 对象中的。然后使用节点（ `QDomNode` ）操作 `doc` 对象，像XML说明，元素，属性，文本等等都被看做是节点，这样就使得操作XML文档变得很简单，我们只需通过转换函数将节点转换成相应的类型，如

```
QDomElement e =n.toElement();
```

在下一节我们将讲述XML文件的创建和写入。

涉及到的源码

第28篇 XML（二）使用DOM创建和操作XML文档

导语

在上一节中我们用手写的方法建立了一个XML文档，并且用DOM的方法对其进行了读取。现在我们使用代码来创建那个XML文档，并且对它实现查找、更新、插入等操作。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录

- 一、创建文档
- 二、读取文档
- 三、添加节点
- 四、查找、删除、更新操作

正文

一、创建文档

1．新建Qt Gui应用，项目名称为`myDom_2`，基类为`QMainWindow`，类名为`MainWindow`。

2．完成后打开`myDom_2.pro`，然后将第一行代码更改为：

```
QT       += core  gui   xml
```

保存该文件。

3．双击`mainwindow.ui`进入设计模式，往界面上添加`Push Button`，`Label`，`Line Edit`，`List Widget`等部件，设计界面如下图所示。



4．完成后，打开 `mainwindow.cpp` 文件，先包含头文件 `#include <QtXml>` ，然后在构造函数中添加如下代码：

```
QFile file("my.xml");
// 只写方式打开，并清空以前的信息
if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate)) return ;

QDomDocument doc;
QDomProcessingInstruction instruction;  //添加处理指令
instruction = doc.createProcessingInstruction("xml","version=\"1.0\"encoding=\"UTF-8\""
);
doc.appendChild(instruction);
QDomElement root = doc.createElement(tr("书库"));
doc.appendChild(root); //添加根元素

// 添加第一个book元素及其子元素
QDomElement book = doc.createElement(tr("图书"));
QDomAttr id = doc.createAttribute(tr("编号"));
QDomElement title = doc.createElement(tr("书名"));
QDomElement author = doc.createElement(tr("作者"));
QDomText text;
id.setValue(tr("1"));
book.setAttributeNode(id);
text = doc.createTextNode(tr("Qt"));
title.appendChild(text);
text = doc.createTextNode(tr("shiming"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);

// 添加第二个book元素及其子元素
book = doc.createElement(tr("图书"));
id = doc.createAttribute(tr("编号"));
title = doc.createElement(tr("书名"));
author = doc.createElement(tr("作者"));
id.setValue(tr("2"));
book.setAttributeNode(id);
text = doc.createTextNode(tr("Linux"));
title.appendChild(text);
text = doc.createTextNode(tr("yafei"));
author.appendChild(text);
book.appendChild(title);
book.appendChild(author);
root.appendChild(book);

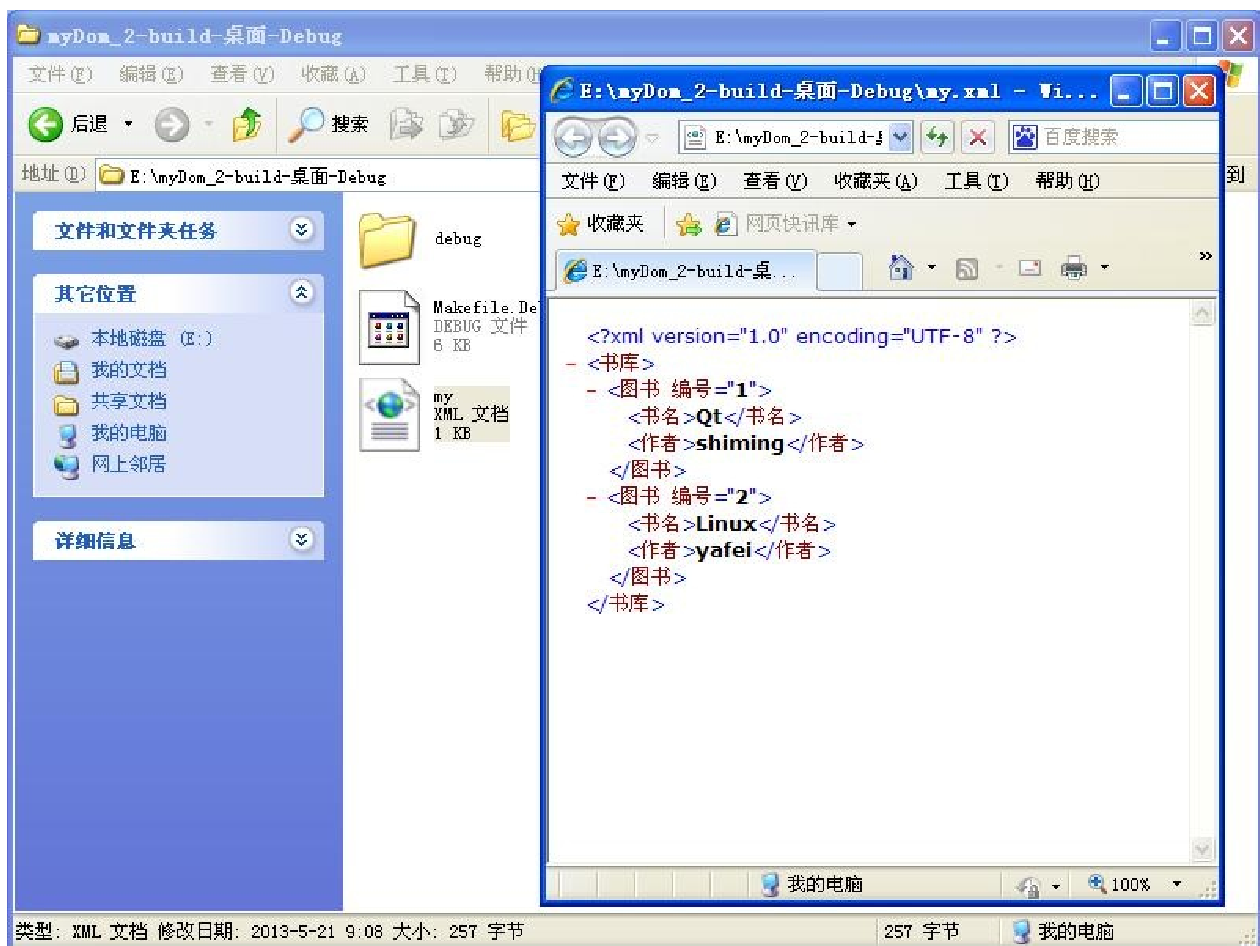
QTextStream out(&file);
doc.save(out,4); // 将文档保存到文件，4为子元素缩进字符数
file.close();
```

这里先使用 `QDomDocument` 类在内存中生成了一棵DOM树，然后调用 `save()` 函数利用 `QTextStream` 文本流将DOM树保存在了文件中。在生成DOM树时主要使用了 `createElement()` 等函数来生成各种节点，然后使用 `appendChild()` 将各个节点依次追加进去。

5．打开 `main.cpp` 文件，先包含头文件： `#include <QTextCodec>` ，然后在 `main()` 函数第一行代码后面添加如下代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("utf8"));
```

6．运行程序，可以看到在构建目录中生成了 `my.xml` 文件，可以双击查看该文件的内容，效果如下图所示。



二、读取文档

下面我们读取整个文档的内容，并显示在 `List Widget` 部件上面，这里用的就是上一节讲到的内容。我们进入“查看全部信息”按钮单击信号槽，更改如下：

```
void MainWindow::on_pushButton_5_clicked()
{
    ui->listWidget->clear(); //先清空显示
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
       file.close();
       return ;
    }
    file.close();

    //返回根节点及其子节点的元素标记名
    QDomElement docElem = doc.documentElement();  //返回根元素
    QDomNode n = docElem.firstChild();   //返回根节点的第一个子节点
    while(!n.isNull())  //如果节点不为空
    {
        if (n.isElement()) //如果节点是元素
       {
           QDomElement e = n.toElement(); //将其转换为元素
           ui->listWidget->addItem(e.tagName()
                                                                    +e.attribut
e(tr("编号")));
           QDomNodeList list = e.childNodes();
           for(int i=0; i<list.count(); i++)
           {
                QDomNode node = list.at(i);
                if(node.isElement())
                   ui->listWidget->addItem("    "
                                                                 +node.toElemen
t().tagName()
                                    +" : "+node.toElement().text());
           }
       }
       n = n.nextSibling();  //下一个兄弟节点
    }
}
```

运行程序，效果如下图所示。



三、添加节点

1・首先在设计模式，把书名和作者标签后面的 `Line Edit` 部件的 `objectName` 分别更改为 `lineEdit_title` 和 `lineEdit_author` 。如下图所示。



2・然后进入添加按钮的单击信号槽，添加如下代码：

```
void MainWindow::on_pushButton_4_clicked()
{
    ui->listWidget->clear(); //我们先清空显示，然后显示“无法添加！”
    ui->listWidget->addItem(tr("无法添加!"));
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
       file.close();
       return;
    }
    file.close();
    QDomElement root = doc.documentElement();
    QDomElement book = doc.createElement(tr("图书"));
    QDomAttr id = doc.createAttribute(tr("编号"));
    QDomElement title = doc.createElement(tr("书名"));
    QDomElement author = doc.createElement(tr("作者"));
    QDomText text;
    // 我们获得了最后一个孩子结点的编号，然后加1，便是新的编号
    QString num = root.lastChild().toElement().attribute(tr("编号"));
    int count = num.toInt() +1;
    id.setValue(QString::number(count));
    book.setAttributeNode(id);
    text = doc.createTextNode(ui->lineEdit_title->text());
    title.appendChild(text);
    text = doc.createTextNode(ui->lineEdit_author->text());
    author.appendChild(text);
    book.appendChild(title);
    book.appendChild(author);
    root.appendChild(book);
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
       return ;
    QTextStream out(&file);
    doc.save(out,4);   //将文档保存到文件，4为子元素缩进字符数
    file.close();
    ui->listWidget->clear(); //最后更改显示为“添加成功！”
    ui->listWidget->addItem(tr("添加成功！"));
}
```

这里先用只读方式打开XML文件，将其读入 `doc` 中，然后关闭。我们将新的节点加入到最后面，并使其“编号”为以前的最后一个节点的编号加1。最后我们再用只写的方式打开XML文件，将修改完的 `doc` 写入其中。运行程序，效果如下图所示。



再次查看全部信息，可以看到新的节点已经添加了，如下图所示。



四、查找、删除、更新操作

因为这三个功能都要先利用“编号”进行查找，所以我们放在一起实现。

1. 首先将界面上“图书编号”后面的 `Line Edit` 部件的 `objectName` 更改为 `lineEdit_id` 。

2. 在 `mainwindow.h` 文件中添加 `public` 类型的函数声明：

```
void doXml(constQString operate);
```

我们使用这个函数来完成三种不同的操作，根据参数来判断不同的操作。

3．然后到 `mainwindow.cpp` 中添加该函数的定义：

```
void MainWindow::doXml(const QString operate)
{
    ui->listWidget->clear();
    ui->listWidget->addItem(tr("没有找到相关内容！"));
    QFile file("my.xml");
    if (!file.open(QIODevice::ReadOnly)) return ;
    QDomDocument doc;
    if (!doc.setContent(&file))
    {
       file.close();
       return ;
    }
       file.close();

    QDomNodeList list = doc.elementsByTagName(tr("图书"));
    // 以标签名进行查找
    for(int i=0; i<list.count(); i++)
    {
       QDomElement e = list.at(i).toElement();
       // 如果元素的"编号"属性值与我们所查的相同
       if(e.attribute(tr("编号")) == ui->lineEdit_id->text())
       {
            // 如果元素的"编号"属性值与我们所查的相同
           if(operate == "delete")  //如果是删除操作
           {
                QDomElement root = doc.documentElement(); //取出根节点
                root.removeChild(list.at(i));  //从根节点上删除该节点
                QFile file("my.xml");     //保存更改
          if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
                                 return ;
                QTextStream out(&file);
                doc.save(out,4);
                file.close();
                ui->listWidget->clear();
                ui->listWidget->addItem(tr("删除成功！"));
           }
           else if(operate == "update")  //如果是更新操作
           {
                QDomNodeList child = list.at(i).childNodes();
                //找到它的所有子节点，就是"书名"和"作者"
                child.at(0).toElement().firstChild().setNodeValue(ui->lineEdit_title->
text());
                //将它子节点的首个子节点（就是文本节点）的内容更新
                child.at(1).toElement().firstChild().setNodeValue(ui->lineEdit_author-
>text());
                QFile file("my.xml");     //保存更改
          if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
                                 return ;
                QTextStream out(&file);
                doc.save(out,4);   //保存文档，4为子元素缩进字符数
                file.close();
                ui->listWidget->clear();
                ui->listWidget->addItem(tr("更新成功！"));
           }
           else if(operate == "find")  //如果是查找操作
           {
                ui->listWidget->clear();
                ui->listWidget->addItem(e.tagName()+e.attribute(tr("编号")));
                QDomNodeList list = e.childNodes();
                for(int i=0; i<list.count(); i++)
               {
                    QDomNode node = list.at(i);
                    if(node.isElement())
                        ui->listWidget->addItem("   "+node.toElement().tagName()
                                 +" : "+node.toElement().text());
                }
           }
       }
```

```
    }
}
```

4．下面我们分别进入“查找”，“删除”，“更新”三个按钮的单击信号槽，更改如下：

```
// 查找
void MainWindow::on_pushButton_clicked()
{
    doXml("find");
}

// 删除
void MainWindow::on_pushButton_2_clicked()
{
      doXml("delete");
}

// 更新void
MainWindow::on_pushButton_3_clicked()
{
      doXml("update");
}
```

下面运行程序，查找操作结果如下图所示。



然后对编号为1的图书进行更新，效果如下图所示。



更新后我们再次查看所有内容。如下图所示。



然后进行删除操作，如下图所示。



删除后再次查询所有内容。效果如下图所示。



结语

通过本节的例子可以看到使用DOM可以很方便的进行XML文档的随机访问，这也是它最大的优点。关于更多更详细的内容可以参考《Qt Creator快速入门》的相关章节。

涉及到的源码

第29篇 XML（三）Qt中的SAX

导语 我们前面讲述了用DOM的方法对XML文档进行操作，DOM实现起来很灵活，但是这样也就使得编程变得复杂了些，而且我们前面也提到过，DOM需要预先把整个XML文档都读入内存，这样就使得它不适合处理较大的文件。下面我们讲述另一种读取XML文档的方法，即SAX 。是的，如果你只想读取并显示整个XML文档，那么SAX是很好的选择，因为它提供了比DOM更简单的接口，并且它不需要将整个XML文档一次性读入内存，这样便可以用来读取较大的文件。我们对SAX不再进行过多的介绍，因为不需要任何基础，你就可以掌握我们下面要讲的内容了。如果大家对SAX有兴趣，可以到网上查找相关资料。

环境：Windows Xp + Qt 4.8.4+QtCreator 2.6.2

目录 一、解析器解析流程 二、使用SAX读取文档

正文

一、解析器解析流程

在Qt的 `QtXml` 模块中提供了一个 `QXmlSimpleReader` 的类，它便是基于SAX的XML解析器。这个解析器是基于事件的，但这些事件由它们自身进行关联，我们并不需要进行设置。我们只需知道，当解析器解析一个XML的元素时，就会执行相应的事件，我们只要重写这些事件处理函数，就能让它按照我们的想法进行解析。

比如要解析下面的元素：

```
<title>Qt</title>
```

解析器会依次调用如下事件处理函数： `startElement()` ， `characters()` ， `endElement()` 。我们可以在 `startElement()` 中获得元素名（如“title”）和属性，在 `characters()` 中获得元素中的文本（如“Qt”），在 `endElement()` 中进行一些结束读取该元素时想要进行的操作。而所有的这些事件处理函数我们都可以通过继承 `QXmlDefaultHandler` 类来重写。

二、使用SAX读取文档

1．新建其他项目分类中的空的Qt项目，项目名称为 `mySAX` 。

2．完成后向项目中添加新的C++类，类名为 `MySAX` ，基类填写 `QXmlDefaultHandler` 。

3．然后再添加一个 `main.cpp` 文件。

4．先打开 `mySAX.pro` 文件，添加一行代码： `QT+= xml` ，然后保存该文件。

5．打开 `mysax.h` 文件，将其内容更改为：

```
#ifndef MYSAX_H
#define MYSAX_H

#include <QXmlDefaultHandler>
class QListWidget;

class MySAX : public QXmlDefaultHandler
{
public:
    MySAX();
    ~MySAX();
    bool readFile(const QString &fileName);
protected:
bool startElement(const QString &namespaceURI,
const QString &localName,
                                          const QString &qName,
const QXmlAttributes &atts);
bool endElement(const QString &namespaceURI,
const QString &localName,
                                    const QString &qName);
    bool characters(const QString &ch);
    bool fatalError(const QXmlParseException &exception);
private:
    QListWidget *list;
    QString currentText;
};

#endif // MYSAX_H
```

这里主要是重新声明了 `QXmlDefaultHandler` 类的 `startElement()` 、 `endElement()` 、 `characters()` 和 `fatalError()` 几个函数， `readFile()` 函数用来读入XML文件， `QListWidget` 部件用来显示解析后的XML文档内容， `currentText` 字符串变量用于暂存字符数据。

6．打开 `mysax.cpp` 文件，将其内容修改如下：

```
#include "mysax.h"
#include <QtXml>
#include <QListWidget>

MySAX::MySAX()
{
    list = new QListWidget;
    list->show();
}

MySAX::~MySAX()
{
    delete list;
}

bool MySAX::readFile(const QString &fileName)
{
    QFile file(fileName);

    // 读取文件内容
    QXmlInputSource inputSource(&file);

    // 建立QXmlSimpleReader对象
    QXmlSimpleReader reader;

    // 设置内容处理器
    reader.setContentHandler(this);

    // 设置错误处理器
```

```
    reader.setErrorHandler(this);

    // 解析文件
    return reader.parse(inputSource);
}

// 已经解析完一个元素的起始标签
bool MySAX::startElement(const QString &namespaceURI,
const QString &localName,
                         const QString &qName,
const QXmlAttributes &atts)
{
    if (qName == "library")
        list->addItem(qName);
    else if (qName == "book")
        list->addItem("    " + qName + atts.value("id"));
    return true;
}

// 已经解析完一块字符数据
bool MySAX::characters(const QString &ch)
{
    currentText = ch;
    return true;
}

// 已经解析完一个元素的结束标签
bool MySAX::endElement(const QString &namespaceURI,
const QString &localName,
                       const QString &qName)
{
    if (qName == "title" || qName == "author")
        list->addItem("        " + qName + " : " + currentText);
    return true;
}

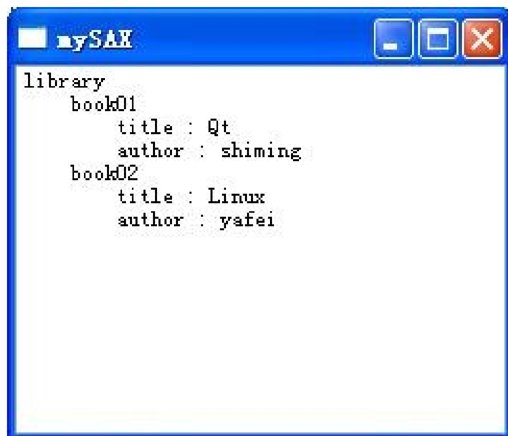
// 错误处理
bool MySAX::fatalError(const QXmlParseException &exception)
{
    qDebug() << exception.message();
    return false;
}
```

这里添加了几个函数的定义。在 `readFile()` 函数中，我们设置了文件的解析过程。Qt中提供了一个简单的XML解析器 `QXmlSimpleReader` ，它是基于SAX的。该解析器需要 `QXmlInputSource` 为其提供数据， `QXmlInputSource` 会使用相应的编码来读取XML文档的数据。在进行解析之前，还需要使用 `setContentHandler()` 来设置事件处理器，使用 `setErrorHandler()` 来设置错误处理器，它们的参数使用了 `this` ，表明使用本类作为处理器，也就是在解析过程中出现的各种事件都会使用本类的 `startElement()` 等事件处理函数来进行处理，而出现错误时会使用本类的 `fatalError()` 函数来处理。最后，调用了 `parse()` 函数来进行解析，该函数会在解析成功时返回 `true` ，否则返回 `false` 。在后面的几个事件处理函数中，就是简单的将数据显示在 `QListWidget` 中。

7．最后打开 `main.cpp` 文件，添加如下内容：

```
#include "mysax.h"
#include <QApplication>
int main(int argc, char* argv[])
{
        QApplication app(argc, argv);
        MySAX sax;
        sax.readFile("../mySAX/my.xml");
        return app.exec();
}
```

8．将前面第27篇建立的 `my.xml` 文件复制到我们的源码目录中，然后运行程序，效果如下图所示。



结语

可以看到使用SAX方法来解析XML文档比使用DOM方法要清晰很多，更重要的是它的效率要高很多，不过SAX方法只适用于读取XML文档。

涉及到的源码

第30篇 XML（四）使用流读写XML

导语

从Qt 4.3开始引入了两个新的类来读取和写入XML文档：`QXmlStreamReader`和`QXmlStreamWriter`。`QXmlStreamReader`类提供了一个快速的解析器通过一个简单的流API来读取格式良好的XML文档，它是作为Qt的SAX解析器的替代品的身份出现的，因为它比SAX解析器更快更方便。`QXmlStreamReader`可以从`QIODevice`或者`QByteArray`中读取数据。流读取器的基本原理就是将XML文档报告为一个记号（tokens）流，这一点与SAX相似，而它们的不同之处在于XML记号被报告的方式。在SAX中，应用程序必须提供处理器（回调函数）来从解析器获得所谓的XML事件；而对于`QXmlStreamReader`，是应用程序代码自身来驱动循环，在需要的时候可以从读取器中一个接一个的拉出记号。这个是通过调用`readNext()`函数实现的，它可以读取下一个记号，然后返回一个记号类型，然后可以使用`isStartElement()`和`text()`等函数来判断这个记号是否包含我们需要的信息。使用这种主动拉取记号的方式的最大的好处就是可以构建递归解析器，也就是可以在不同的函数或者类中来处理XML文档中的不同记号。

环境：Windows Xp + Qt 4.8.4+Qt Creator2.6.2

目录

正文

一、解析XML文档

1．新建Qt控制台应用，项目名称为`myXmlStream`，完成后将`myXmlStream.pro`文件的第一行代码更改为：

```
QT       += core  xml
```

然后保存该文件。

2．然后打开`main.cpp`文件，将内容更改如下：

```
#include <QCoreApplication>
#include <QFile>
#include <QXmlStreamReader>
#include <QXmlStreamWriter>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("../myXmlStream/my.xml");
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        qDebug()<<"Error: cannot open file";
        return 1;
    }

    QXmlStreamReader reader;

    // 设置文件，这时会将流设置为初始状态
    reader.setDevice(&file);

    // 如果没有读到文档结尾，而且没有出现错误
    while (!reader.atEnd()) {
        // 读取下一个记号，它返回记号的类型
        QXmlStreamReader::TokenType type = reader.readNext();

        // 下面便根据记号的类型来进行不同的输出
        if (type == QXmlStreamReader::StartDocument)
            qDebug() << reader.documentEncoding()
                                 << reader.documentVersion();
        if (type == QXmlStreamReader::StartElement) {
            qDebug() << "<" << reader.name() << ">";
            if (reader.attributes().hasAttribute("id"))
                qDebug() << reader.attributes().value("id");
        }
        if (type == QXmlStreamReader::EndElement)
            qDebug() << "</" << reader.name() << ">";
        if (type == QXmlStreamReader::Characters
                             && !reader.isWhitespace())
            qDebug() << reader.text();
    }

    // 如果读取过程中出现错误，那么输出错误信息
    if (reader.hasError()) {
        qDebug() << "error: " << reader.errorString();
    }

    file.close();

    return a.exec();
}
```

可以看到流读取器就是在一个循环中通过使用 `readNext()` 来不断读取记号的，这里可以对不同的记号和不同的内容进行不同的处理，既可以在本函数中进行，也可以在其他函数或者其他类中进行。可以将前面生成的 `my.xml` 文件复制到源码目录，然后运行程序，查看效果。

二、写入XML文档

与 `QXmlStreamReader` 对应的是 `QXmlStreamWriter` ，它通过一个简单的流API提供了一个XML写入器。 `QXmlStreamWriter` 的使用是十分简单的，只需要调用相应的记号的写入函数来写入相关数据即可。

将前面主函数的内容更改为：

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile file("../myXmlStream/my2.xml");
    if (!file.open(QFile::WriteOnly | QFile::Text))
    {
       qDebug() << "Error: cannot open file";
       return 1;
    }
    QXmlStreamWriter stream(&file);
    stream.setAutoFormatting(true);
    stream.writeStartDocument();
    stream.writeStartElement("bookmark");
    stream.writeAttribute("href", "http://qt.nokia.com/");
    stream.writeTextElement("title", "Qt Home");
    stream.writeEndElement();
    stream.writeEndDocument();
    file.close();
    qDebug() << "write finished!";
    return a.exec();
}
```

这里使用了 `setAutoFormatting(true)` 函数来自动设置格式，这样会自动换行和添加缩进。然后使用了 `writeStartDocument()` ，该函数会自动添加首行的XML说明（即 `<?xmlversion="1.0" encoding="UTF-8"?>` ），添加元素可以使用 `writeStartElement()` ，不过，这里要注意，一定要在元素的属性、文本等添加完成后，使用 `writeEndElement()` 来关闭前一个打开的元素。在最后使用 `writeEndDocument()` 来完成文档的写入。现在大家可以运行程序了，这时会在项目目录中生成一个XML文档。

结语

数据库和XML在很多程序中都经常用到，它们的使用也总是和数据的显示联系起来，所以学习好数据库的知识也是很重要的，它们可以说是密不可分的。相关内容，大家也可以参考《Qt Creator快速入门》的相关章节以及《Qt 及Qt Quick开发实战精解》的数据管理系统的例子，里面同时应用了数据库和XML。

网络篇

第31篇 网络（一）Qt网络编程简介

导语

从这一节开始我们讲述Qt网络应用方面的编程知识。在开始这部分知识的学习之前，大家最好已经拥有了一定的网络知识和Qt的编程基础。在后面的教程中我们不会对一个常用的网络名词进行详细的解释，对于不太了解的地方，大家可以参考相关书籍。

不过，大家也没有必要非得先去学习网络专业知识，而后再学习本部分内容，因为Qt提供了简单明了的接口函数，使得这里并不需要了解太多专业的知识。看完教程后，你也许会发现，自己虽然不懂网络，但却可以编写网络应用程序了。

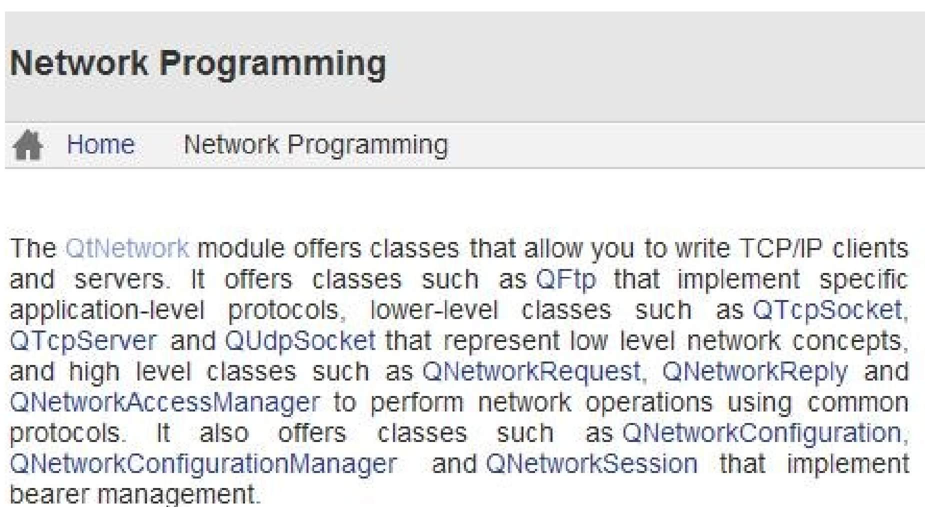
环境：Windows Xp + Qt 4.8.5+Qt Creator 2.8.0

目录

正文

一、了解Qt中的网络编程

1．首先我们打开Qt Creator，进入帮助模式，然后在索引中查找：Network Programming关键字。这里详细介绍了Qt中网络编程的相关内容。如下图所示。



Network Programming

Home　Network Programming

The QtNetwork module offers classes that allow you to write TCP/IP clients and servers. It offers classes such as QFtp that implement specific application-level protocols, lower-level classes such as QTcpSocket, QTcpServer and QUdpSocket that represent low level network concepts, and high level classes such as QNetworkRequest, QNetworkReply and QNetworkAccessManager to perform network operations using common protocols. It also offers classes such as QNetworkConfiguration, QNetworkConfigurationManager and QNetworkSession that implement bearer management.

Qt提供了`QtNetwork`模块来进行网络编程。该模块提供了诸如`QFtp`等类来实现特定的应用层协议；有较低层次的类，例如`QTcpSocket`、`QTcpServer`和`QUdpSocket`等来表示低层的网络概念；还有高层次的类，例如`QNetworkRequest`、`QNetworkReply`和`QNetworkAccessManager`使用相同的协议来执行网络操作；也提供了`QNetworkConfiguration`、`QNetworkConfigurationManager`和`QNetworkSession`等类来实现负载管理。

2．在文档的后面提供了Qt中用于网络编程的类的列表。如下图所示。

Qt's Classes for Network Programming

The following classes provide support for network programming in Qt.

QAbstractSocket	The base functionality common to all socket types
QAuthenticator	Authentication object
QFtp	Implementation of the client side of FTP protocol
QHostAddress	IP address
QHostInfo	Static functions for host name lookups
QHttpMultiPart	Resembles a MIME multipart message to be sent over HTTP
QHttpPart	Holds a body part to be used inside a HTTP multipart MIME message
QNetworkAccessManager	Allows the application to send network requests and receive replies

3．如果大家以前就使用过Qt进行网络部分编程，或者看过其他教材上相关内容，你可能会问，这里怎么没有了`QHttp`类。我们现在搜索`QHttp`关键字，其内容如下。

QHttp Class Reference

Home Modules QtNetwork QHttp

The QHttp class provide

protocol. More...

```
#include <QHttp>
```

This class is obsolete. It is provided to keep old source code working. We strongly advise against using it in new code.

可以看到这里有一个警告：

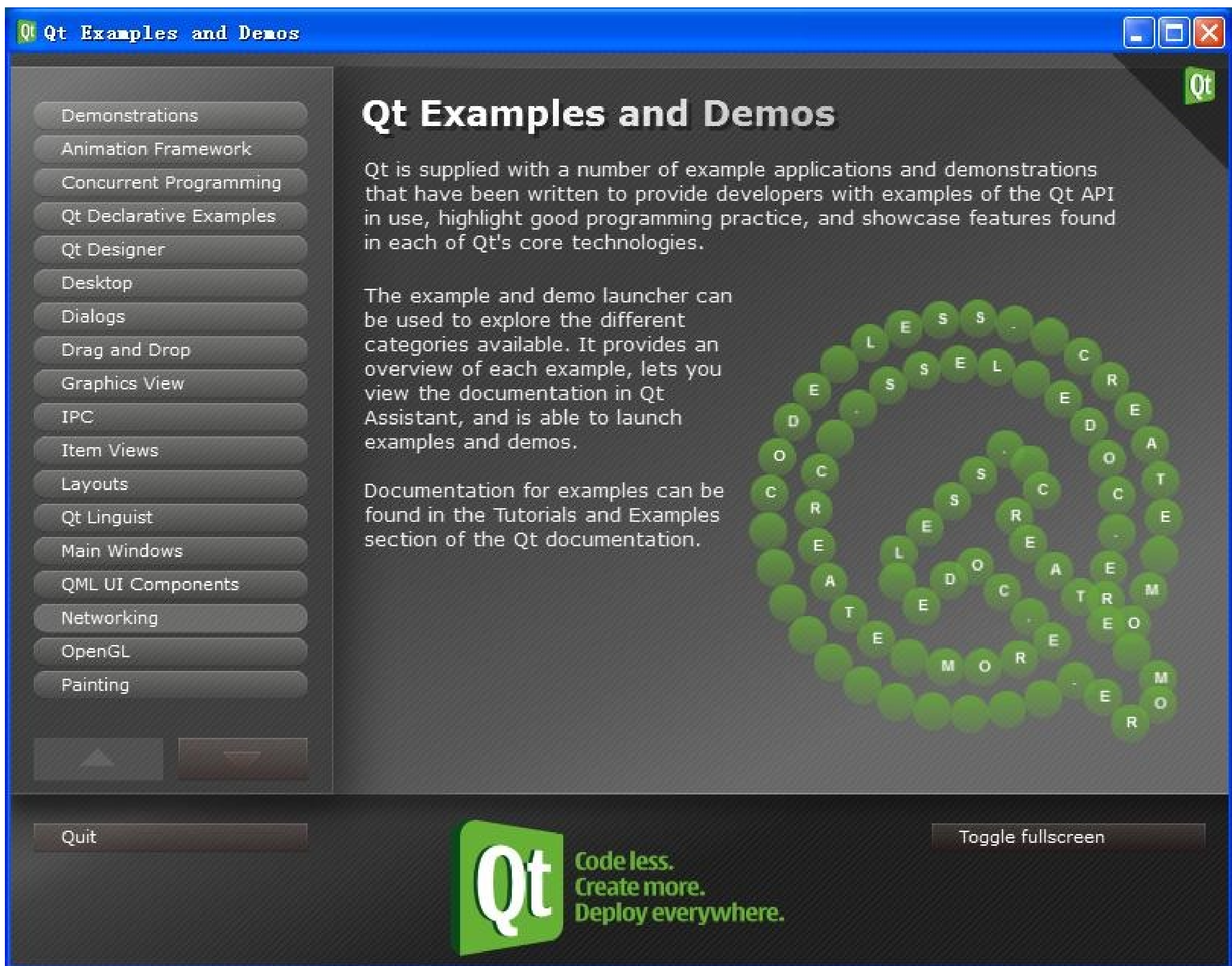
This class is obsolete. It is provided to keep old source code working. We strongly advise against using it in new code.

大概意思是：这个类是过时的。它的提供只是为了保证旧的源代码。我们强烈建议在新代码中不要使用它。

所以在我们的教程中不会再讲解这个类，对于HTTP部分的编程，我们使用 `QNetworkAccessManager` 类和 `QNetworkReply` 类 。

二、查看网络部分的例子

对于网络编程部分的例子，可以通过Qt自带的演示程序查看。就是开始菜单中Qt安装目录下Example and demos程序，网络编程例子在Networking分类中。如下图所示。



进入Networking分类，如下图所示。



最后要说明的是：如果要使用QtNetwork模块中的类，需要在项目文件中添加QT+= network一行代码。

结语

后面的教程中我们将对Qt网络编程部分的知识点分别进行讲解，对Qt中网络编程内容有了初步了解以后，我们就开始下一步的学习吧。

第32篇 网络（二）HTTP

导语

HTTP（HyperText Transfer Protocol，超文本传输协议）是一个客户端和服务器端请求和应答的标准。在Qt的网络模块中提供了网络访问接口来实现HTTP编程。网络访问接口是执行一般的网络操作的类的集合，该接口在特定的操作和使用的协议（例如，通过HTTP进行获取和发送数据）上提供了一个抽象层，只为外部暴露出了类、函数和信号。

上一节中我们已经提到过了，现在Qt中使用 `QNetworkAccessManager` 类和 `QNetworkReply` 类来进行HTTP的编程。网络请求由 `QNetworkRequest` 类来表示，它也作为与请求有关的信息（例如，任何头信息和使用加密）的容器。在创建请求对象时指定的URL决定了请求使用的协议，目前支持HTTP、FTP和本地文件URLs的上传和下载。 `QNetworkAccessManager` 类用来协调网络操作，每当一个请求创建后，该类用来调度它，并发射信号来报告进度。该类还协调 `cookies` 的使用，身份验证请求，及其代理的使用等。对于网络请求的应答使用 `QNetworkReply` 类表示，它会在请求被完成调度时由 `QNetworkAccessManager` 来创建。 `QNetworkReply` 提供的信号可以用来单独的监视每一个应答。

下面我们先讲解一个简单下载网页的例子，然后将其扩展为可以下载任何文件。

环境：Windows Xp + Qt 4.8.5+Qt Creator 2.8.0

目录

正文

一、简单的网页浏览功能

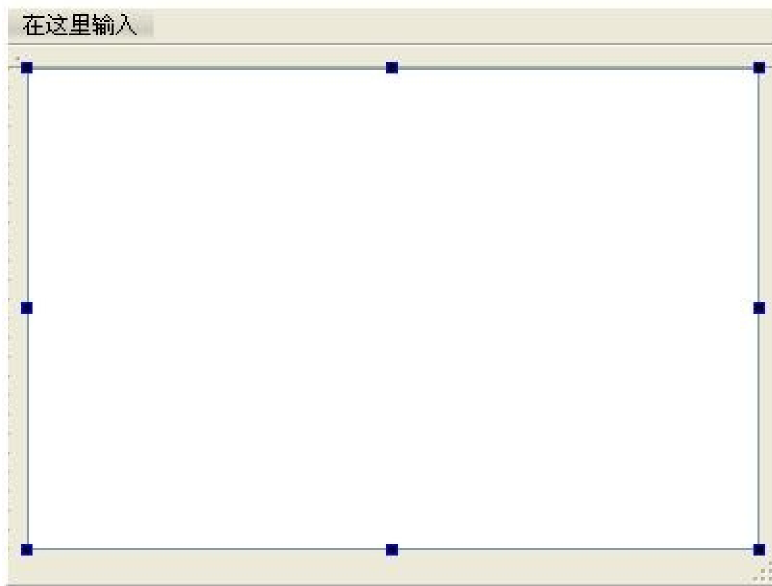
1．新建Qt Gui应用，项目名称为 `http` ，基类使用默认的 `QMainWindow` 即可，类名为 `MainWindow` 。

2．完成后打开 `http.pro` 文件，然后添加下面一行代码来使用网络模块：

```
QT  += network
```

然后保存该文件。

3．下面打开 `mainwindow.ui` 文件进入设计模式，向界面上添加一个 `Text Browser` 部件。效果如下图所示。



4．打开 `mainwindow.h` 文件，先包含头文件： `#include <QtNetwork>`

然后添加一个 `private` 私有对象定义： `QNetworkAccessManager *manager;`

最后添加一个私有槽声明：

```
private slots:
    void replyFinished(QNetworkReply *);
```

5．下面到 `mainwindow.cpp` 文件中，先在构造函数中添加如下代码：

```
manager = new QNetworkAccessManager(this);
connect(manager, SIGNAL(finished(QNetworkReply*)),
this,SLOT(replyFinished(QNetworkReply*)));
manager->get(QNetworkRequest(QUrl("http://www.qter.org")));
```

这里先创建了一个 `QNetworkAccessManager` 类的实例，它用来发送网络请求和接收应答。然后关联了管理器的 `finished()` 信号和我们自定义的槽，每当网络应答结束时都会发射这个信号。最后使用了 `get()` 函数来发送一个网络请求，网络请求使用 `QNetworkRequest` 类表示， `get()` 函数返回一个 `QNetworkReply` 对象。除了 `get()` 函数，管理器还提供了发送HTTP POST请求的 `post()` 函数。

6．下面添加槽的定义：

```
void MainWindow::replyFinished(QNetworkReply *reply)
{
    QTextCodec *codec = QTextCodec::codecForName("utf8");
    QString all = codec->toUnicode(reply->readAll());
    ui->textBrowser->setText(all);
    reply->deleteLater();
}
```

因为 `QNetworkReply` 继承自 `QIODevice` 类，所以可以操作一般的I/O设备一样来操作该类。这里使用了 `readAll()` 函数来读取所有的应答数据，为了正常显示中文，使用了 `QTextCodec` 类来转换编码。在完成数据的读取后，需要使用 `deleteLater()` 来删除 `replay` 对象。

7．因为这里使用了 `QtextCodec` 类，所以还需要在 `mainwindow.cpp` 文件中包含头文件

```
#include <QTextCodec>
```

下面运行程序，效果如下图所示。



这里再将整个过程简答叙述一遍：上面实现了最简单的应用HTTP协议下载网页的功能。 `QNetworkAccessManager` 类用于发送网络请求和接受回复，具体来说，它是用 `QNetworkRequest` 类来管理请求， `QNetworkReply` 类进行接收回复，并对数据进行处理。

在上面的代码中，我们使用了下面的代码来发送请求：

```
manager->get(QNetworkRequest(QUrl("http://www.qter.org")));
```

它返回一个 `QNetworkReply` 对象，这个后面再讲。我们只需知道只要发送请求成功，它就会下载数据。而当数据下载完成后， `manager` 会发出 `finished()` 信号，我们关联了该信号：

```
connect(manager, SIGNAL(finished(QNetworkReply*)),
this,SLOT(replyFinished(QNetworkReply*)));
```

也就是说，当下载数据结束时，就会执行`replyFinished()`函数。在这个函数中我们对接收的数据进行处理：

```
QTextCodec *codec = QTextCodec::codecForName("utf8");
QString all = codec->toUnicode(reply->readAll());
ui->textBrowser->setText(all);
```

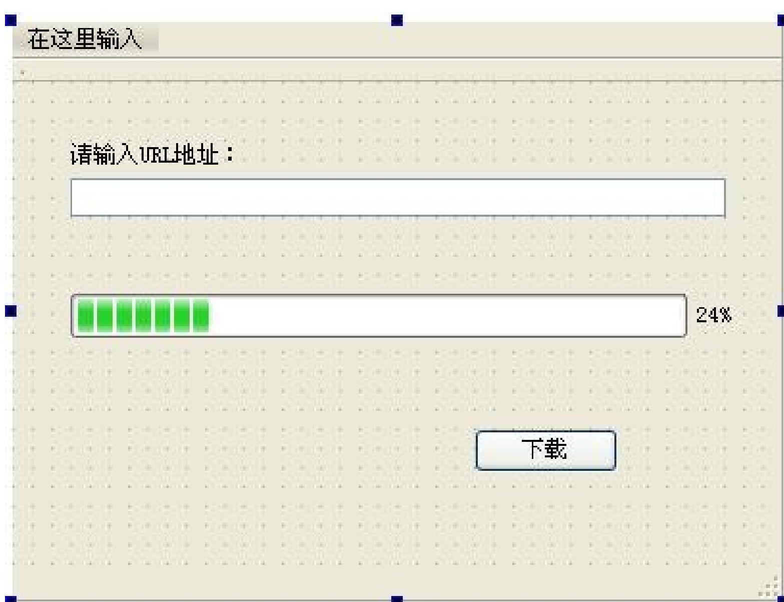
这里，为了能显示下载的网页中的中文，我们使用了`QTextCodec`类对象，应用utf8编码。使用`reply->readAll()`函数就可以将下载的所有数据读出。然后，我们在`textBrowser`中将数据显示出来。当`reply`对象已经完成了它的功能时，我们需要将它释放，就是最后一条代码：

```
reply->deleteLater();
```

二、实现下载文件功能

通过上面的例子可以看到，Qt中编写基于HTTP协议的程序是十分简单的，只有十几行代码。不过，一般我们下载文件都想要看到下载进度。下面我们就更改上面的程序，让它可以下载任意的文件，并且显示下载进度。

1. 进入设计模式，删除以前的`Text Browser`部件，然后拖入一个`Line Edit`，一个`Label`，一个`Progress Bar`和一个`Push Button`，设计界面如下图所示。



2. 在写代码之前，我们先介绍一下整个程序执行的流程：开始我们先让进度条隐藏。当我们在Line Edit中输入下载地址，点击下载按钮后，我们应用输入的下载地址，获得文件名，在磁盘上新建一个文件，用于保存下载的数据，然后进行链接，并显示进度条。在下载过程中，我们将每次获得的数据都写入文件中，并更新进度条，在接收完文件后，我们重新隐藏进度条，并做一些清理工作。根据这个思路，我们开始代码的编写。

3．到 `mainwindow.h` 中，首先添加 `public` 函数声明：

```
void startRequest(QUrl url); //请求链接
```

然后添加几个private变量和对象定义：

```
QNetworkReply *reply;
QUrl url;   //存储网络地址
QFile *file;  //文件指针
```

最后到 `private slots` 中，删除前面的 `replyFinished(QNetworkReply *)` 槽声明，并添加如下代码：

```
private slots:
    void on_pushButton_clicked();  //下载按钮的单击事件槽函数
    void httpFinished();  //完成下载后的处理
    void httpReadyRead();  //接收到数据时的处理
    void updateDataReadProgress(qint64, qint64); //更新进度条
```

4．下面到 `mainwindow.cpp` 文件中，将前面在构造函数中添加的内容删除，然后添加如下代码：

```
manager = new QNetworkAccessManager(this);
ui->progressBar->hide();
```

我们在构造函数中先隐藏进度条。等开始下载时再显示它。

5．下面将前面程序中添加的 `replyFinished()` 函数的定义删除，然后添加新的函数的定义。先添加网络请求函数的实现：

```
void MainWindow::startRequest(QUrl url)
{
    reply = manager->get(QNetworkRequest(url));
    connect(reply, SIGNAL(readyRead()), this, SLOT(httpReadyRead()));

    connect(reply, SIGNAL(downloadProgress(qint64, qint64)),
            this, SLOT(updateDataReadProgress(qint64, qint64)));

    connect(reply, SIGNAL(finished()), this, SLOT(httpFinished()));
}
```

这里使用了 `get()` 函数来发送网络请求，然后进行了 `QNetworkReply` 对象的几个信号和自定义槽的关联。其中 `readyRead()` 信号继承自 `QIODevice` 类，每当有新的数据可以读取时，都会发射该信号；每当网络请求的下载进度更新时都会发射 `downloadProgress()` 信号，它用来更新进度条；每当应答处理结束时，都会发射 `finished()` 信号，该信号与前面程序中 `QNetworkAccessManager` 类的 `finished()` 信号作用相同，只不过是发送者不同，参数也不同而已。下面添加几个槽的定义。

```
void MainWindow::httpReadyRead()
{
    if (file) file->write(reply->readAll());
}
```

这里先判断是否创建了文件，如果是，则读取返回的所有数据，然后写入到文件。该文件是在后面的“下载”按钮单击信号槽中创建并打开的。

```
void MainWindow::updateDataReadProgress(qint64 bytesRead, qint64 totalBytes)
{
    ui->progressBar->setMaximum(totalBytes);
    ui->progressBar->setValue(bytesRead);
}
```

这里设置了一下进度条的最大值和当前值。

```
void MainWindow::httpFinished()
{
    ui->progressBar->hide();
    file->flush();
    file->close();
    reply->deleteLater();
    reply = 0;
    delete file;
    file = 0;
}
```

当完成下载后，重新隐藏进度条，然后删除 `reply` 和 `file` 对象。下面是“下载”按钮的单击信号的槽：

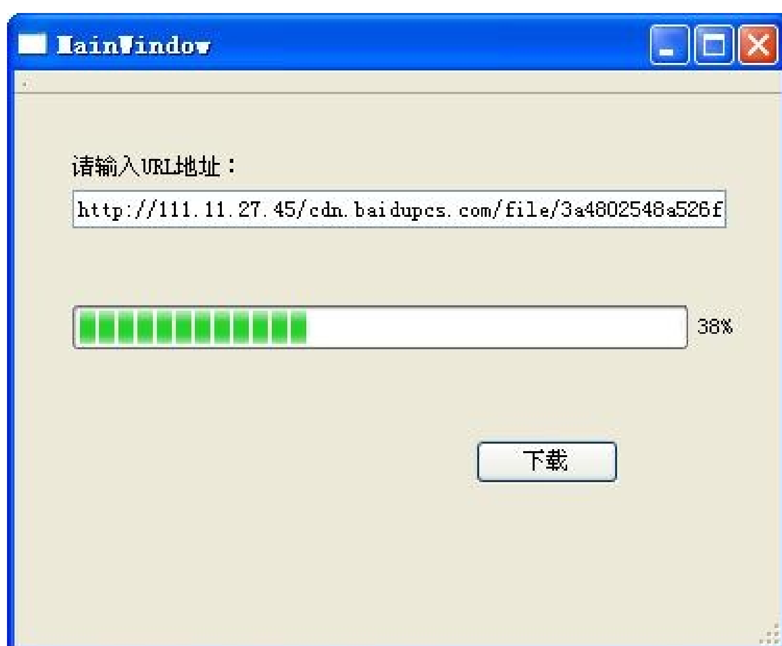
```
void MainWindow::on_pushButton_clicked()
{
    url = ui->lineEdit->text();
    QFileInfo info(url.path());
    QString fileName(info.fileName());
    if (fileName.isEmpty()) fileName = "index.html";
    file = new QFile(fileName);
    if(!file->open(QIODevice::WriteOnly))
    {
        qDebug() << "file open error";
        delete file;
        file = 0;
        return;
    }
    startRequest(url);
    ui->progressBar->setValue(0);
    ui->progressBar->show();
}
```

这里使用要下载的文件名创建了本地文件，然后使用输入的 `url` 进行了网络请求，并显示进度条。

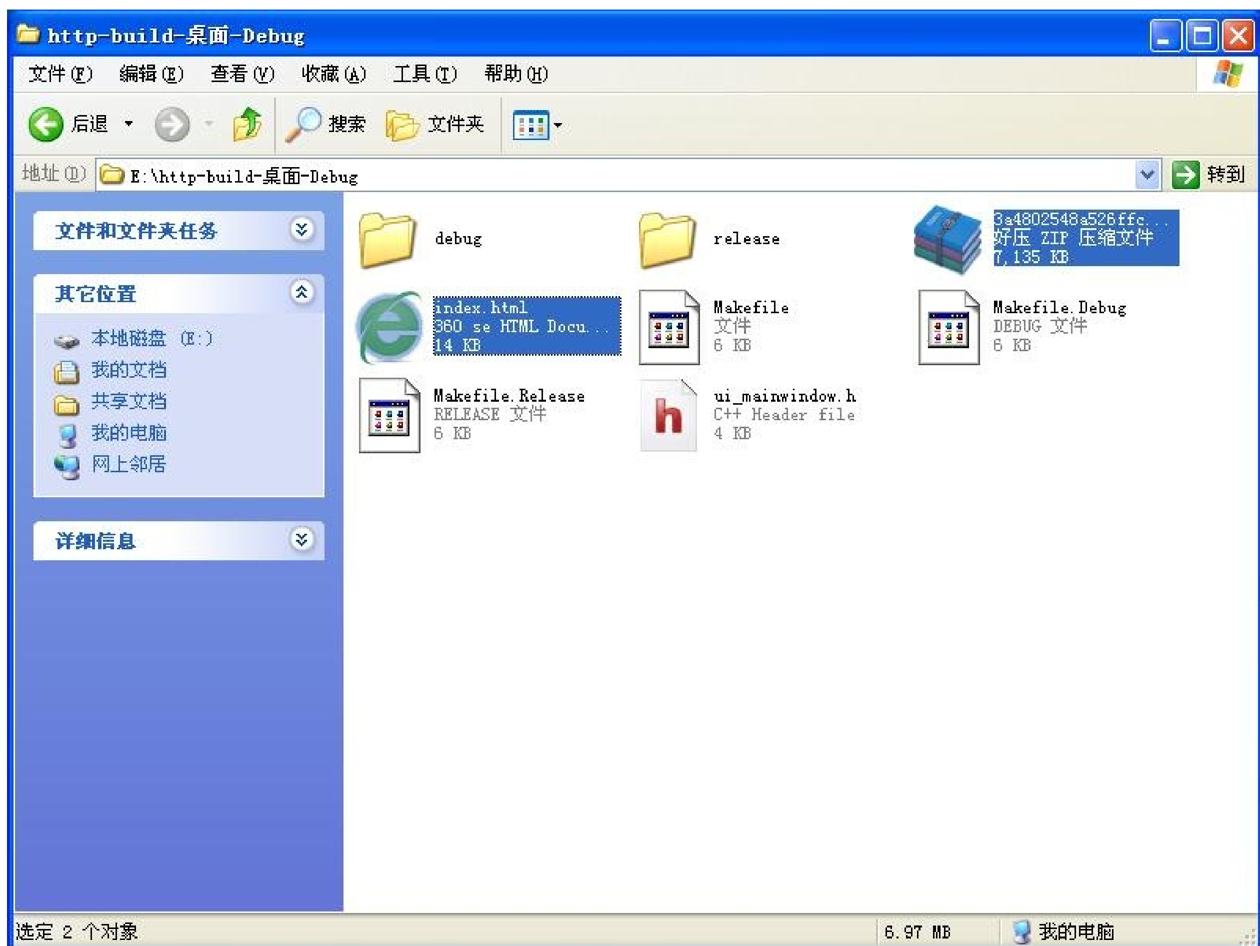
6．下面运行程序，我们先输入 `www.yafeilinux.com` 的网址，下面一个网页。效果如下图所示。



完成后，可以尝试输入一个文件的下载地址，比如这里输入了《Qt Creator快速入门》一书在百度网盘上的地址，效果如下图所示。



7．最后，可以去项目编译生成的文件目录中查看下载的文件（我这里是 `E:\http-build-桌面-Debug` ），可以看到下载的文件，如下图所示。



结语

HTTP应用的内容就讲到这里，可以看到它是很容易的，也不需要大家了解太多的HTTP的原理知识。关于相关的类的其他使用，也可以查看其帮助文档。在上面的例子中，我们只是为了讲解知识，所以程序不是很完善，对于一个真正的工程，还是需要注意更多其他细节的，大家可以查看一下Qt演示程序HTTP Client的源代码。

涉及到的源码

第33篇 网络（三）FTP（一）

导语

上一节我们讲述了HTTP的编程，这一节讲述与其及其相似的FTP的编程。FTP即FileTransfer Protocol，也就是文件传输协议。FTP的主要作用，就是让用户连接上一个远程计算机，查看远程计算机有哪些文件，然后把文件从远程计算机上拷贝到本地计算机，或者把本地计算机的文件送到远程计算机上。

环境：Windows Xp + Qt 4.8.5+QtCreator 2.8.0

目 录

- 一、简介
- 二、实现简单的文件下载

正文

一、简介

在Qt中，我们可以使用上一节讲述的 `QNetworkAccessManager` 和 `QNetworkReply` 类来进行FTP程序的编写，因为它们用起来很简单。但是，对于较复杂的FTP操作，Qt还提供了 `QFtp` 类，利用这个类，我们很容易写出一个FTP客户端程序。下面我们先在帮助中查看这个类。

QFtp Class Reference

Home　Modules　QtNetwork　QFtp

The QFtp class provides an implem

```
#include <QFtp>
```

nherits: QObject.

- List of all members, including inherited members
- Qt 3 support members

Public Types

在QFtp中，所有的操作都对应一个特定的函数，我们可以称它们为命令。如 `connectToHost()` 连接到服务器命令，`login()` 登录命令，`get()` 下载命令，`mkdir()` 新建目录命令等。因为 `QFtp` 类以异步方式工作，所以所有的这些函数都不是阻塞函数。也就是说，如果一个操作不能立即执行，那么这个函数就会直接返回，直到程序控制权返回Qt事件循环后才真正执行，它们不会影响界面的显示。

所有的命令都返回一个 `int` 型的编号，使用这个编号让我们可以跟踪这个命令，查看其执行状态。当每条命令开始执行时，都会发出 `commandStarted()` 信号，当该命令执行结束时，会发出 `commandFinished()` 信号。我们可以利用这两个信号和命令的编号来获取命令的执行状态。当然，如果不想执行每条命令都要记下它的编号，也可以使用 `currentCommand()` 来获取现在执行的命令，其返回值与命令的对应关系如下图。

enum QFtp:: Command

This enum is used as the return value for the currentCommand() function. Thi
directory view when a list() command is started; in this case you can simply chec

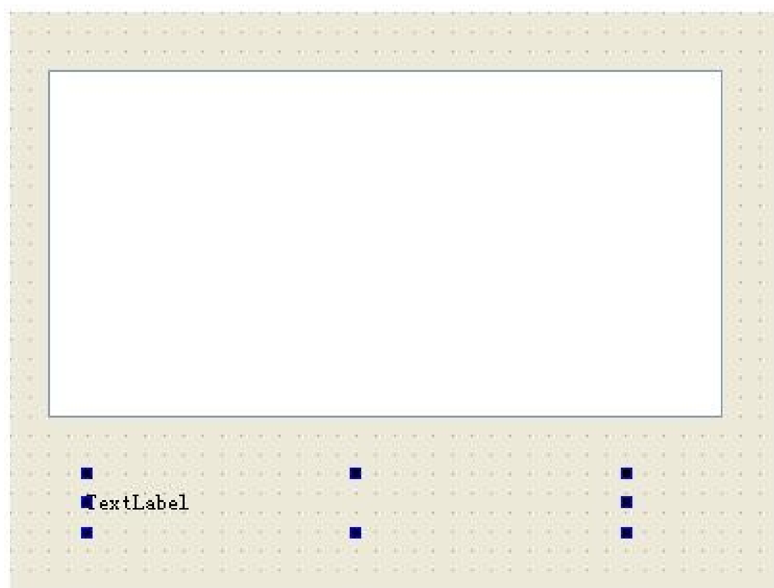
Constant	Value	Description
QFtp::None	0	No command is being executed.
QFtp::SetTransferMode	1	set the transfer mode.
QFtp::SetProxy	2	switch proxying on or off.
QFtp::ConnectToHost	3	connectToHost() is being executed.
QFtp::Login	4	login() is being executed.
QFtp::Close	5	close() is being executed.
QFtp::List	6	list() is being executed.
QFtp::Cd	7	cd() is being executed.
QFtp::Get	8	get() is being executed.
QFtp::Put	9	put() is being executed.
QFtp::Remove	10	remove() is being executed.
QFtp::Mkdir	11	mkdir() is being executed.
QFtp::Rmdir	12	rmdir() is being executed.
QFtp::Rename	13	rename() is being executed.
QFtp::RawCommand	14	rawCommand() is being executed.

二、实现简单的文件下载

下面我们先看一个简单的FTP客户端的例子，然后对它进行扩展。在这个例子中我们从FTP服务器上下载一个文件并显示出来。

1．我们新建Qt Gui应用。 项目名次为 `myFtp` ，基类选择 `QWidget` ，类名保持 `Widget` 即可。完成后打开 `muFtp.pro` 文件，在上面添加一行： `QT += network` ，然后保存该文件。

2．修改 `widget.ui` 文件。 在其中添加一个 `TextBrowser` 和一个 `Label` ，效果如下。



3. 在 `main.cpp` 中进行修改。

为了在程序中可以使用中文，我们在 `main.cpp` 中添加头文件 `#include <QTextCodec>`

并在 `main()` 函数中添加代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
```

4. 在 `widget.h` 中进行修改。

先添加头文件：`#include <QFtp>`

再在 `private` 中定义对象：`QFtp *ftp;`

添加私有槽函数：

```
private slots:
    void ftpCommandStarted(int);
void ftpCommandFinished(int,bool);
```

5. 在 `widget.cpp` 中进行更改。

（1）在构造函数中添加代码：

```
ftp = new QFtp(this);
ftp->connectToHost("ftp.qt-project.org"); //连接到服务器
ftp->login();    //登录
ftp->cd("qt/source");    //跳转到“qt”目录下的source目录中
ftp->get("INSTALL");  //下载“INSTALL”文件
ftp->close();    //关闭连接

// 当每条命令开始执行时发出相应的信号
connect(ftp,SIGNAL(commandStarted(int)),
        this,SLOT(ftpCommandStarted(int)));

// 当每条命令执行结束时发出相应的信号
connect(ftp,SIGNAL(commandFinished(int,bool)),
        this,SLOT(ftpCommandFinished(int,bool)));
```

我们在构造函数里执行了几个FTP的操作，登录站点，并下载了一个文件。然后又关联了两个信号和槽，用来跟踪命令的执行情况。

（2）实现槽函数：

```
void Widget::ftpCommandStarted(int)
{
    if(ftp->currentCommand() == QFtp::ConnectToHost){
       ui->label->setText(tr("正在连接到服务器..."));
    }
    if (ftp->currentCommand() == QFtp::Login){
       ui->label->setText(tr("正在登录..."));
    }
    if (ftp->currentCommand() == QFtp::Get){
       ui->label->setText(tr("正在下载..."));
    }
    else if (ftp->currentCommand() == QFtp::Close){
       ui->label->setText(tr("正在关闭连接..."));
    }
}
```

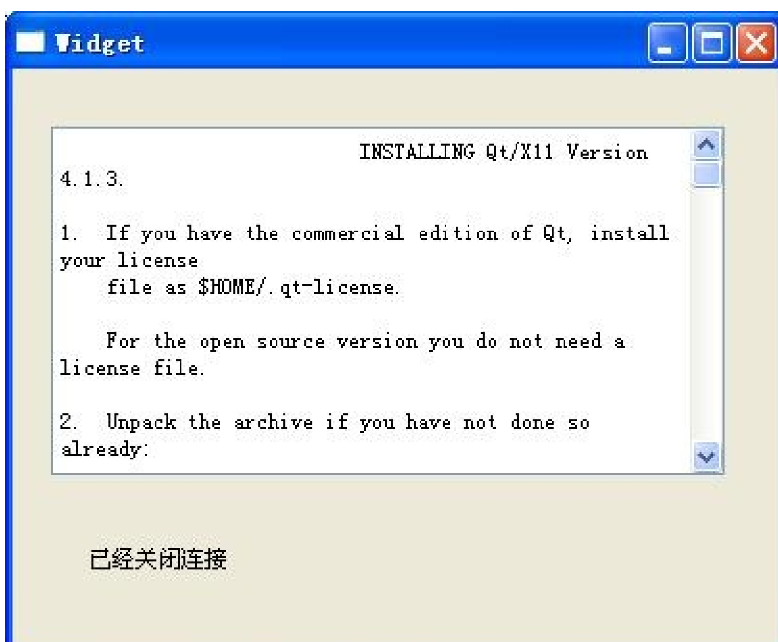
每当命令执行时，都会执行 `ftpCommandStarted()` 函数，它有一个参数 `int id` ，这个 `id` 就是调用命令时返回的 `id` ，如 `int loginID= ftp->login();` 这时，我们就可以用 `if(id == loginID)` 来判断执行的是否是 `login()` 函数。但是，我们不想为每个命令都设置一个变量来存储其返回值，所以，我们这里使用了 `ftp->currentCommand()` ，它也能获取当前执行的命令的类型。在这个函数里我们让开始不同的命令时显示不同的状态信息。

```
void Widget::ftpCommandFinished(int,bool error)
{
    if(ftp->currentCommand() == QFtp::ConnectToHost){
       if(error)
           ui->label->setText(tr("连接服务器出现错误：%1")
                                .arg(ftp->errorString()));
       else ui->label->setText(tr("连接到服务器成功"));
    }
    if (ftp->currentCommand() == QFtp::Login){
       if(error)
           ui->label->setText(tr("登录出现错误：%1")
                                .arg(ftp->errorString()));
       else ui->label->setText(tr("登录成功"));
    }
    if (ftp->currentCommand() == QFtp::Get){
       if(error)
           ui->label->setText(tr("下载出现错误：%1")
                                .arg(ftp->errorString()));
       else {
           ui->label->setText(tr("已经完成下载"));
           ui->textBrowser->setText(ftp->readAll());
       }
    }
    else if (ftp->currentCommand() == QFtp::Close){
       ui->label->setText(tr("已经关闭连接"));
    }
}
```

这个函数与 `ftpCommandStarted()` 函数相似，但是，它是在一个命令执行结束时执行的。它有两个参数，第一个 `intid` ，就是调用命令时返回的编号，我们在上面已经讲过了。第二个是 `bool error` ，它标志现在执行的命令是否出现了错误。如果出现了错误，那么 `error` 为 `true` ，否则为 `false` 。我们可以利用它来输出错误信息。在这个函数中，我们在完成一条命令时显示不同的状态信息，并显示可能的出错信息。

在 `if (ftp->currentCommand() == QFtp::Get)` 中，也就是已经完成下载时，我们让 `textBrowser` 显示下载的信息。

6．运行程序，效果如下。



7．出错演示。

下面我们演示一下出错时的情况。

将构造函数中的代码 `ftp->login();` 改为 `ftp->login("tom","123456");`

这时我们再运行程序：



可以看到，它输出了错误信息，指明了错误的指令和出错的内容。其实我们设置的这个错误，也是想告诉大家，在FTP中如果没有设置用户名和密码，那么默认的用户名应该是 `anonymous` ，这时密码可以任意填写，而使用其他用户名是会出错的。

结语

在下一节中，我们将会对这个程序进行扩展，让它可以浏览服务器上的所有文件，并进行下载。

涉及的源码下载

第34篇 网络（四）FTP（二）

导语

前面讲述了一个最简单的FTP客户端程序的编写，这一节我们将这个程序进行扩展，使其可以浏览并能下载服务器上的所有文件。

环境：Windows Xp + Qt 4.8.5+QtCreator 2.8.0

目 录

正文

一、修改界面

我们删除了`TextBrowser`，加入了几个`Label`，`Line Edit`，`Push Button`部件，一个`Tree Widge`t及一个`Progress Bar`部件。然后我们对其中几个部件做如下更改。

（1）将“FTP服务器”标签后的`Line Edit`的`objectName`属性改为`ftpServerLineEdit`，其`text`属性改为`ftp.qt-project.org`。

（2）将“用户名”标签后的`Line Edit`的`objectName`属性改为`userNameLineEdit`，其`text`属性改为`anonymous`，将其`toolTip`属性改为“默认用户名请使用：anonymous ，此时密码任意。”

（3）将“密码”标签后的`Line Edit`的`objectName`属性改为`passWordLineEdit`，其`text`属性改为`123456`，将其`echoMode`属性改为`Password`。

（4）将“连接”按钮的`objectName`属性改为`connectButton`。

（5）将“返回上一级目录”按钮的`objectName`属性改为`cdToParentButton`。

（6）将“下载”按钮`的objectName`属性改为`downloadButton`。

（7）将`Tree Widget`的`objectName`属性改为`fileList`，然后在`Tree Widget`部件上单击鼠标右键，选择`Edit Items`菜单，添加列属性如下。



最终界面如图所示：



下面我们的程序中，就是实现在用户填写完相关信息后，按下“连接”按钮，就可以连接到FTP服务器，并在 `TreeWidget` 中显示服务器上的所有文件，我们可以按下“下载”按钮来下载选中的文件，并使用进度条显示下载进度。

二、功能实现

1．更改 `widget.h` 文件。

（1）添加头文件 `#include <QtGui>`

（2）在 `private` 中添加变量：

```
QHash<QString, bool> isDirectory; //用来存储一个路径是否为目录的信息
QString currentPath; //用来存储现在的路径
QFile *file;
```

（3）添加槽：

```
private slots:
void on_downloadButton_clicked();
void on_cdToParentButton_clicked();
void on_connectButton_clicked();
void ftpCommandFinished(int,bool);
void ftpCommandStarted(int);
void updateDataTransferProgress(qint64,qint64 );//更新进度条
//将服务器上的文件添加到Tree Widget中
void addToList(const QUrlInfo &urlInfo);
void processItem(QTreeWidgetItem*,int);//双击一个目录时显示其内容
```

2. 更改 `widget.cpp` 的内容。

（1）实现“连接”按钮的单击事件槽。

```
void Widget::on_connectButton_clicked()   //连接按钮
{
    ui->fileList->clear();
    currentPath.clear();
    isDirectory.clear();

    ftp = new QFtp(this);
    connect(ftp,SIGNAL(commandStarted(int)),
this,SLOT(ftpCommandStarted(int)));
    connect(ftp,SIGNAL(commandFinished(int,bool)),
           this,SLOT(ftpCommandFinished(int,bool)));
    connect(ftp,SIGNAL(listInfo(QUrlInfo)),
this,SLOT(addToList(QUrlInfo)));
    connect(ftp,SIGNAL(dataTransferProgress(qint64,qint64)),
           this,SLOT(updateDataTransferProgress(qint64,qint64)));

    QString ftpServer = ui->ftpServerLineEdit->text();
    QString userName = ui->userNameLineEdit->text();
    QString passWord = ui->passWordLineEdit->text();
    ftp->connectToHost(ftpServer,21); //连接到服务器,默认端口号是21
    ftp->login(userName,passWord);   //登录
}
```

我们在“连接”按钮的单击事件槽函数中新建了 `ftp` 对象，然后关联了相关的信号和槽。这里的 `listInfo()` 信号由 `ftp->list()` 函数发射，它将在登录命令完成时调用，下面我们提到。而 `dataTransferProgress()` 信号在数据传输时自动发射。最后我们从界面上获得服务器地址，用户名和密码等信息，并以它们为参数执行连接和登录命令。

（2）更改 `ftpCommandFinished()` 函数。

我们在相应位置做更改。

首先，在登录命令完成时，我们调用list()函数：

```
ui->label->setText(tr("登录成功"));
ftp->list();   //发射listInfo()信号，显示文件列表
然后，在下载命令完成时，我们使下载按钮可用，并关闭打开的文件。
ui->label->setText(tr("已经完成下载"));
ui->downloadButton->setEnabled(true);
file->close();
delete file;
```

最后再添加一个 `if` 语句，处理 `list` 命令完成时的情况：

```
if (ftp->currentCommand() == QFtp::List){
       if (isDirectory.isEmpty())
       { //如果目录为空,显示“empty”
           ui->fileList->addTopLevelItem(
                  new QTreeWidgetItem(QStringList()<< tr("<empty>")));
           ui->fileList->setEnabled(false);
           ui->label->setText(tr("该目录为空"));
       }
}
```

我们在 `list` 命令完成时，判断文件列表是否为空，如果为空，就让 `Tree Widget` 不可用，并显示“empty”条目。

（3）添加文件列表函数的内容如下。

```
void Widget::addToList(const QUrlInfo &urlInfo)  //添加文件列表
{
    QTreeWidgetItem *item = new QTreeWidgetItem;
    item->setText(0, urlInfo.name());
    item->setText(1, QString::number(urlInfo.size()));
    item->setText(2, urlInfo.owner());
    item->setText(3, urlInfo.group());
    item->setText(4, urlInfo.lastModified().toString("MMM dd yyyy"));

    QPixmap pixmap(urlInfo.isDir() ? "../myFtp2/dir.png" : "../myFtp2/file.png");
    item->setIcon(0, pixmap);

    isDirectory[urlInfo.name()] = urlInfo.isDir();
    //存储该路径是否为目录的信息
    ui->fileList->addTopLevelItem(item);
    if (!ui->fileList->currentItem()) {
       ui->fileList->setCurrentItem(ui->fileList->topLevelItem(0));
       ui->fileList->setEnabled(true);
    }
}
```

当 `ftp->list()` 函数执行时会发射 `listInfo()` 信号，此时就会执行 `addToList()` 函数，在这里我们将文件信息显示在 `Tree Widget` 上，并在 `isDirectory` 中存储该文件的路径及其是否为目录的信息。为了使文件与目录进行区分，我们使用了不同的图标 `file.png` 和 `dir.png` 来表示它们，这两个图标放在了工程文件夹中。

（4）将构造函数的内容更改如下。

```
{
    ui->setupUi(this);
    ui->progressBar->setValue(0);
    //鼠标双击列表中的目录时，我们进入该目录
    connect(ui->fileList,SIGNAL(itemActivated(QTreeWidgetItem*,int)),
           this,SLOT(processItem(QTreeWidgetItem*,int)));
}
```

这里我们只是让进度条的值为0，然后关联了 `Tree Widget` 的一个信号 `itemActivated()` 。当鼠标双击一个条目时，发射该信号，我们在槽函数中判断该条目是否为目录，如果是则进入该目录。

（5） `processItem()` 函数的实现如下。

```
void Widget::processItem(QTreeWidgetItem* item,int)  //打开一个目录
{
    QString name = item->text(0);
    if (isDirectory.value(name)) {  //如果这个文件是个目录，则打开
       ui->fileList->clear();
       isDirectory.clear();
       currentPath += '/';
       currentPath += name;
       ftp->cd(name);
       ftp->list();
       ui->cdToParentButton->setEnabled(true);
    }
}
```

（6）“返回上一级目录”按钮的单击事件槽函数如下。

```
void Widget::on_cdToParentButton_clicked()  //返回上级目录按钮
{
    ui->fileList->clear();
    isDirectory.clear();
    currentPath = currentPath.left(currentPath.lastIndexOf('/'));
    if (currentPath.isEmpty()) {
       ui->cdToParentButton->setEnabled(false);
       ftp->cd("/");
    } else {
       ftp->cd(currentPath);
    }
    ftp->list();
}
```

在返回上一级目录时，我们取当前路径的最后一个 `/` 之前的部分，如果此时路径为空了，我们就让“返回上一级目录”按钮不可用。

（7）“下载”按钮单击事件槽函数如下。

```
void Widget::on_downloadButton_clicked()  //下载按钮
{
    QString fileName = ui->fileList->currentItem()->text(0);
    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly))
    {
       delete file;
       return;
}
//下载按钮不可用，等下载完成后才可用
    ui->downloadButton->setEnabled(false);    ftp->get(ui->fileList->currentItem()->te
xt(0), file);
}
```

在这里我们获取了当前项目的文件名，然后新建文件，使用 `get()` 命令下载服务器上的文件到我们新建的文件中。

（8）更新进度条函数内容如下。

```
void Widget::updateDataTransferProgress( //进度条
                  qint64 readBytes,qint64 totalBytes)
{
    ui->progressBar->setMaximum(totalBytes);
    ui->progressBar->setValue(readBytes);
}
```

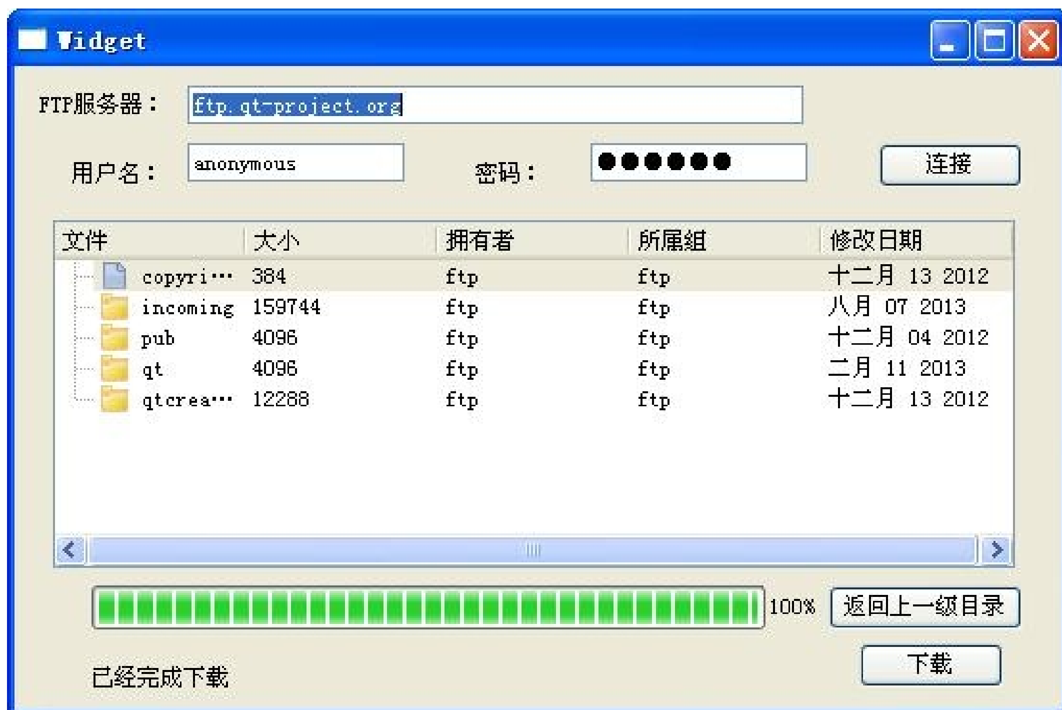
3．流程说明。

整个程序的流程就和我们实现函数的顺序一样。用户在界面上输入服务器的相关信息，然后我们利用这些信息进行连接并登录服务器，等登录服务器成功时，我们列出服务器上所有的文件。对于一个目录，我们可以进入其中，并返回上一级目录，我们可以下载文件，并显示下载的进度。

对于 `ftp` 的操作，全部由那些命令和信号来完成，我们只需要调用相应的命令，并在其发出信号时，进行对应的处理就可以了。而对于文件的显示，则是视图部分的知识了。

4．运行程序，效果如下图所示。





结语

最后需要说明的是，因为为了更好的讲解知识，使得程序简单化，所以我们省去了很多细节上的处理，如果需要，你可以自己添加。比如断开连接和取消下载，你都可以使用

`ftp->abort()` 函数。你也可以参考Qt自带的Ftp Example例子。对于其他操作，比如上传等，你可以根据需要添加。

FTP的相关编程就讲到到这里。

涉及到的源码下载

第35篇 网络（五）获取本机网络信息

导语

前面讲完了HTTP和FTP，下面本来该讲解UDP和TCP了。不过，在讲解它们之前，我们先在这一节里讲解一个以后要经常用到的名词，那就是IP地址。

对于IP地址，其实，会上网的人都应该听说过它。如果你实在很不属性，那么简单的说：IP即InternetProtocol（网络之间互联的协议），协议就是规则，地球人都用一样的规则，所以我们可以访问全球任何的网站；而IP地址就是你联网时分配给你机子的一个地址。如果把网络比喻成地图，那IP地址就像地图上的经纬度一样，它确定了你的主机在网络中的位置。其实知道我们以后要用IP地址来代表网络中的一台计算机就够了。（^_^不一定科学但是很直白的表述）

下面我们就讲解如何获取自己电脑的IP地址以及其他网络信息。这一节中，我们会涉及到网络模块（ `QtNetworkModule` ）中的 `QHostInfo` ， `QHostAddress` ， `QNetworkInterface` 和 `QNetworkAddressEntry` 等几个类。下面是详细内容。

环境：Windows Xp + Qt 4.8.5+Qt Creator2.8.0

目录

正文

一、使用 `QHostInfo` 获取主机名和IP地址

我们新建Qt Gui应用，项目名为 `myIP` ，基类选择 `QWidget` ，类名保持 `Widget` 不变。完成后先打开 `myIP.pro` 文件，添加一行代码： `QT += network` ，然后保存该文件。下面打开 `widget.h` 文件添加头文件包含： `#include <QtNetwork>`

（1）获取主机名。

我们在 `widget.cpp` 文件中的构造函数中添加代码：

```
QString localHostName = QHostInfo::localHostName();
qDebug() <<"localHostName: "<<localHostName;
```

这里我们使用了 `QHostInfo` 类的 `localHostName` 类来获取本机的计算机名称。

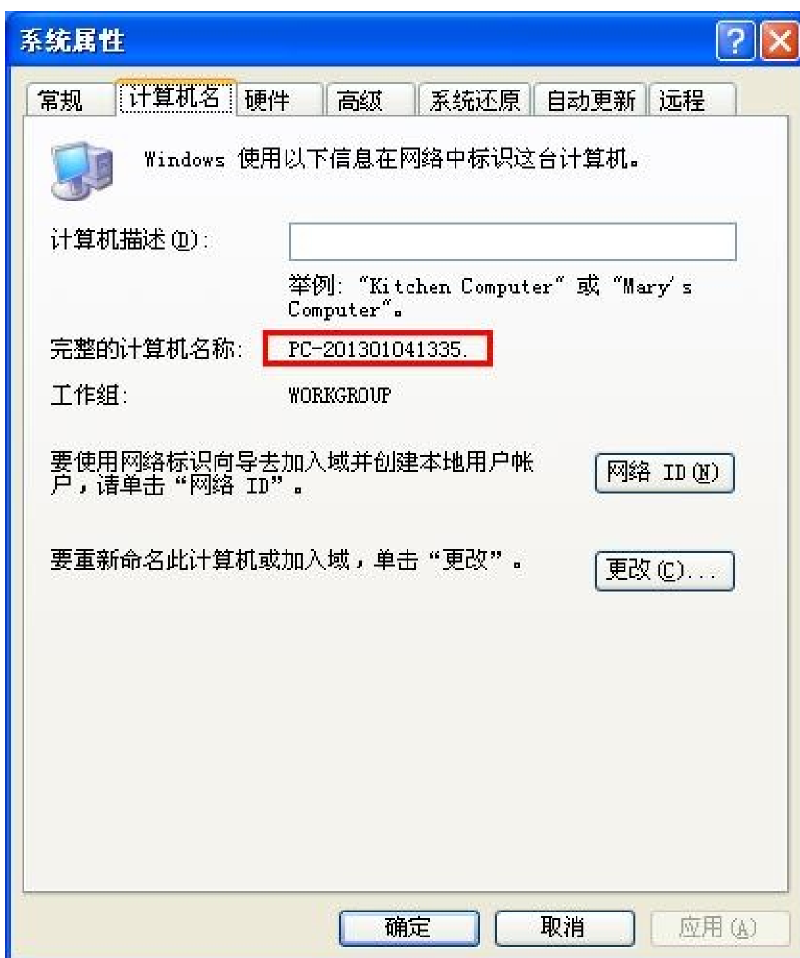
运行程序，在下面的应用程序输出栏里的信息如下：

应用程序输出

myIP

```
E:\build-myIP-Desktop_Qt_4_8_5-Debug\debug\
localHostName:  "PC-201301041335"
```

可以看到，这里获取了计算机名。我们可以在桌面上“我的电脑”图标上点击鼠标右键，然后选择“属性”菜单，查看“计算机名”一项，和这里输出结果是一样的，如下图。

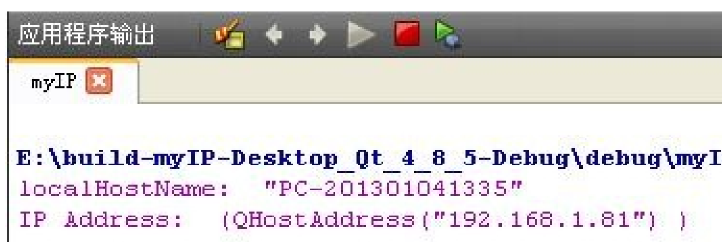


（2）获取本机的IP地址。

我们继续在构造函数中添加代码：

```
QHostInfo info = QHostInfo::fromName(localHostName);
qDebug() <<"IP Address: "<<info.addresses();
```

调用 `QHostInfo` 类的 `fromName()` 函数，使用上面获得的主机名为参数，来获取本机的信息。然后再利用 `QHostInfo` 类的 `addresses()` 函数，获取本机的所有IP地址信息。运行程序，输出信息如下：



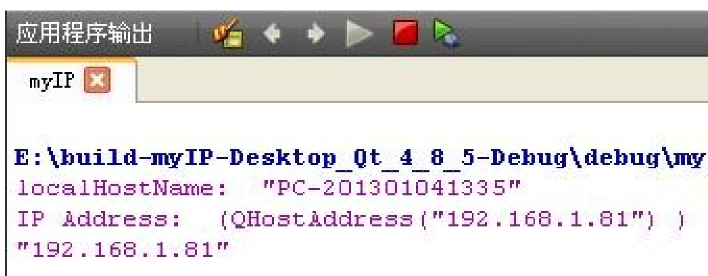
在我这里只有一条IP地址。但是，在其他系统上，可能出现多条IP地址，其中可能包含了IPv4和IPv6的地址，一般我们需要使用IPv4的地址，所以我们可以只输出IPv4的地址。

我们继续添加代码：

```
foreach(QHostAddress address,info.addresses())
{
     if(address.protocol() == QAbstractSocket::IPv4Protocol)
        qDebug() << address.toString();
}
```

因为IP地址由 `QHostAddress` 类来管理，所以我们可以使用该类来获取一条IP地址，然后使用该类的 `protocol()` 函数来判断其是否为IPv4地址。如果是IPv6地址，可以使用 `QAbstractSocket::IPv6Protocol` 来判断。最后我们将IP地址以 `QString` 类型输出。

我们以后要使用的IP地址都是用这个方法获得的，所以这个一定要掌握。运行效果如下：



（3）以主机名获取IP地址。

在上面讲述了用本机的计算机名获取本机的IP地址。其实 `QHostInfo` 类也可以用来获取任意主机名的IP地址，如一个网站的IP地址。在这里我们可以使用 `lookupHost()` 函数。它是基于信号和槽的，一旦查找到了IP地址，就会触发槽函数。具体用法如下。

我们在 `widget.h` 文件中添加一个私有槽函数：

```
private slots:
void lookedUp(const QHostInfo &host);
```

然后在 `widget.cpp` 中的构造函数中先将上面添加的代码全部注释（可以通过选中所有代码，然后按下 `Ctrl+/` 快捷键来注释代码），然后添加以下代码：

```
QHostInfo::lookupHost("www.qter.org",
                      this,SLOT(lookedUp(QHostInfo)));
```

这里我们查询Qter开源社区的IP地址，如果查找到，就会执行我们的 `lookedUp()` 函数。

在 `widget.cpp` 中添加 `lookedUp()` 函数的实现代码：

```
void Widget::lookedUp(const QHostInfo &host)
{
    qDebug() << host.addresses().first().toString();
}
```

这里我们只是简单地输出第一个IP地址。输出信息如下：



其实，我们也可以使用 `lookupHost()` 函数，通过输入IP地址反向查找主机名，只需要将上面代码中的 `www.qter.org` 换成一个IP地址就可以了，如果你有兴趣可以研究一下，不过返回的结果可能不是你想象中的那样。

可以看到 `QHostInfo` 类的作用：通过主机名来查找IP地址，或者通过IP地址来反向查找主机名。

二、通过 `QNetworkInterface` 类来获取本机的IP地址和网络接口信息

`QNetworkInterface` 类提供了程序所运行时的主机的IP地址和网络接口信息的列表。在每一个网络接口信息中都包含了0个或多个IP地址，而每一个IP地址又包含了和它相关的子网掩码和广播地址，它们三者被封装在一个 `QNetworkAddressEntry` 对象中。网络接口信息中也提供了硬件地址信息。我们将 `widge.cpp` 构造函数中以前添加的代码注释掉，然后添加以下代码。

```
//获取所有网络接口的列表
QList<QNetworkInterface> list = QNetworkInterface::allInterfaces();
foreach(QNetworkInterface interface,list) //遍历每一个网络接口
{
    qDebug() << "Device: "<<interface.name(); //设备名
//硬件地址
qDebug() << "HardwareAddress: "<<interface.hardwareAddress();

//获取IP地址条目列表，每个条目中包含一个IP地址，
//一个子网掩码和一个广播地址
   QList<QNetworkAddressEntry> entryList= interface.addressEntries();
    foreach(QNetworkAddressEntry entry,entryList)//遍历每个IP地址条目
    {
       qDebug()<<"IP Address: "<<entry.ip().toString(); //IP地址
       qDebug()<<"Netmask: "<<entry.netmask().toString(); //子网掩码
     qDebug()<<"Broadcast: "<<entry.broadcast().toString();//广播地址
    }
}
```

这里我们获取了本机的网络设备的相关信息。运行程序，输出如下：

应用程序输出

myIP

```
E:\build-myIP-Desktop_Qt_4_8_5-Debug\debug\myIP.e:
Device:  "{DF24512B-A723-4330-A6F0-395ED2F26A19}"
HardwareAddress:  "74:27:EA:22:12:75"
IP Address:  "192.168.1.81"
Netmask:  "255.255.255.0"
Broadcast:  "192.168.1.255"
Device:  "MS TCP Loopback interface"
HardwareAddress:  ""
IP Address:  "127.0.0.1"
Netmask:  ""
Broadcast:  ""
```

其实，如果我们只想利用`QNetworkInterface`类来获取IP地址，那么就没必要像上面那样复杂，这个类提供了一个便捷的函数`allAddresses()`来获取IP地址，例如：

```
QString address = QNetworkInterface::allAddresses().first().toString();
```

结语

在这一节中我们学习了如何来查找本机网络设备的相关信息。其实，以后最常用的还是其中获取IP地址的方法。我们以后可以利用一个函数来获取IP地址：

```
QString Widget::getIP()  //获取ip地址
{
    QList<QHostAddress> list = QNetworkInterface::allAddresses();
    foreach (QHostAddress address, list)
    {
       //我们使用IPv4地址
       if(address.protocol() == QAbstractSocket::IPv4Protocol)
           return address.toString();
    }
    return 0;
}
```

这一节就讲到这里，在下面的几节中我们将利用IP地址进行UDP和TCP的编程。

涉及的源码

第36篇 网络（六）UDP

导语

这一节讲述UDP编程的知识。UDP（UserDatagram Protocol即用户数据报协议）是一个轻量级的，不可靠的，面向数据报的无连接协议。对于UDP我们不再进行过多介绍，如果你对UDP不是很了解，而且不知道它有什么用，那么这里就举个简单的例子：我们现在几乎每个人都使用的腾讯QQ，其聊天时就是使用UDP协议进行消息发送的。就像QQ那样，当有很多用户，发送的大部分都是短消息，要求能及时响应，并且对安全性要求不是很高的情况下使用UDP协议。

在Qt中提供了 `QUdpSocket` 类来进行UDP数据报（datagrams）的发送和接收。这里我们还要了解一个名词Socket，也就是常说的“套接字”。 Socket简单地说，就是一个IP地址加一个port端口。因为我们要传输数据，就要知道往哪个机子上传送，而IP地址确定了一台主机，但是这台机子上可能运行着各种各样的网络程序，我们要往哪个程序中发送呢？这时就要使用一个端口来指定UDP程序。所以说，Socket指明了数据报传输的路径。

下面我们将编写两个程序，一个用来发送数据报，可以叫做客户端；另一个用来接收数据报，可以叫做服务器端，它们均应用UDP协议。这样也就构成了所谓的C/S（客户端/服务器）编程模型。我们会在编写程序的过程中讲解一些相关的网络知识。

环境：Windows Xp + Qt 4.8.5+QtCreator 2.8.0

目录

正文

一、发送端（客户端）

1．新建Qt Gui应用。 项目名为 `udpSender` ，基类选择 `QWidget` ，类名为 `Widget` 。完成后在 `udpSender.pro` 文件中添加一行代码： `QT += network` ，并保存该文件。

2．在 `widget.ui` 文件中，往界面上添加一个 `Push Button` ，更改其显示文本为“开始广播”，然后进入其单击事件槽函数。

3．我们在 `widget.h` 文件中更改。

添加头文件： `#include <QtNetwork>`

添加private私有对象： `QUdpSocket *sender;`

4．我们在 `widget.cpp` 中进行更改。

在构造函数中添加： `sender = new QUdpSocket(this);`

更改“开始广播”按钮的单击事件槽函数：

```
void Widget::on_pushButton_clicked() // 开始广播
{
    QByteArray datagram = "hello world!";
    sender->writeDatagram(datagram.data(),datagram.size(),
                          QHostAddress::Broadcast,45454);
}
```

这里定义了一个 `QByteArray` 类型的数据报 `datagram` ，其内容为“hello world!”。然后我们使用 `QUdpSocket` 类的 `writeDatagram()` 函数来发送数据报，这个函数有四个参数，分别是数据报的内容，数据报的大小，主机地址和端口号。对于数据报的大小，它根据平台的不同而不同，但是这里建议不要超过512字节。这里使用了广播地址 `QHostAddress::Broadcast` ，这样就可以同时给网络中所有的主机发送数据报了。对于端口号，它是可以随意指定的，但是一般1024以下的端口号通常属于保留端口号，所以我们最好使用大于1024的端口，最大为65535。我们这里使用了45454这个端口号，一定要注意，在下面要讲的服务器程序中，也要使用相同的端口号。

5．发送端就这么简单，下面可以先运行程序。

二、接收端（服务器端）

1．新建Qt Gui 应用

工程名为 `udpReceiver` ，基类选择QWidget，类名为 `Widget` 。完成后在 `udpSender.pro` 文件中添加一行代码： `QT += network` ，并保存该文件。

此时工程文件列表中应包含两个项目，如下图。



2、我们在 `udpReceiver` 项目中的 `widget.ui` 文件中，向界面上添加一个 `Label` 部件，更改其显示文本为“等待接收数据！”，效果如下。

3、我们在 `udpReceiver` 工程中的 `widget.h` 文件中更改。

添加头文件： `#include <QtNetwork>`

添加private私有对象： `QUdpSocket *receiver;`

添加私有槽函数：

```
private slots:
void processPendingDatagram();
```

4、我们在 `udpReceiver` 工程中的 `widget.cpp` 文件中更改。

在构造函数中：

```
receiver = new QUdpSocket(this);
receiver->bind(45454,QUdpSocket::ShareAddress);
connect(receiver,SIGNAL(readyRead()),
this,SLOT(processPendingDatagram()));
```

我们在构造函数中将 `receiver` 绑定到45454端口，这个端口就是上面发送端设置的端口，二者必须一样才能保证接收到数据报。这里使用了绑定模式 `QUdpSocket::ShareAddress` ，它表明其他服务也可以绑定到这个端口上。因为当 `receiver` 发现有数据报到达时就会发出 `readyRead()` 信号，所以将其和数据报处理函数相关联。

数据报处理槽函数实现如下：

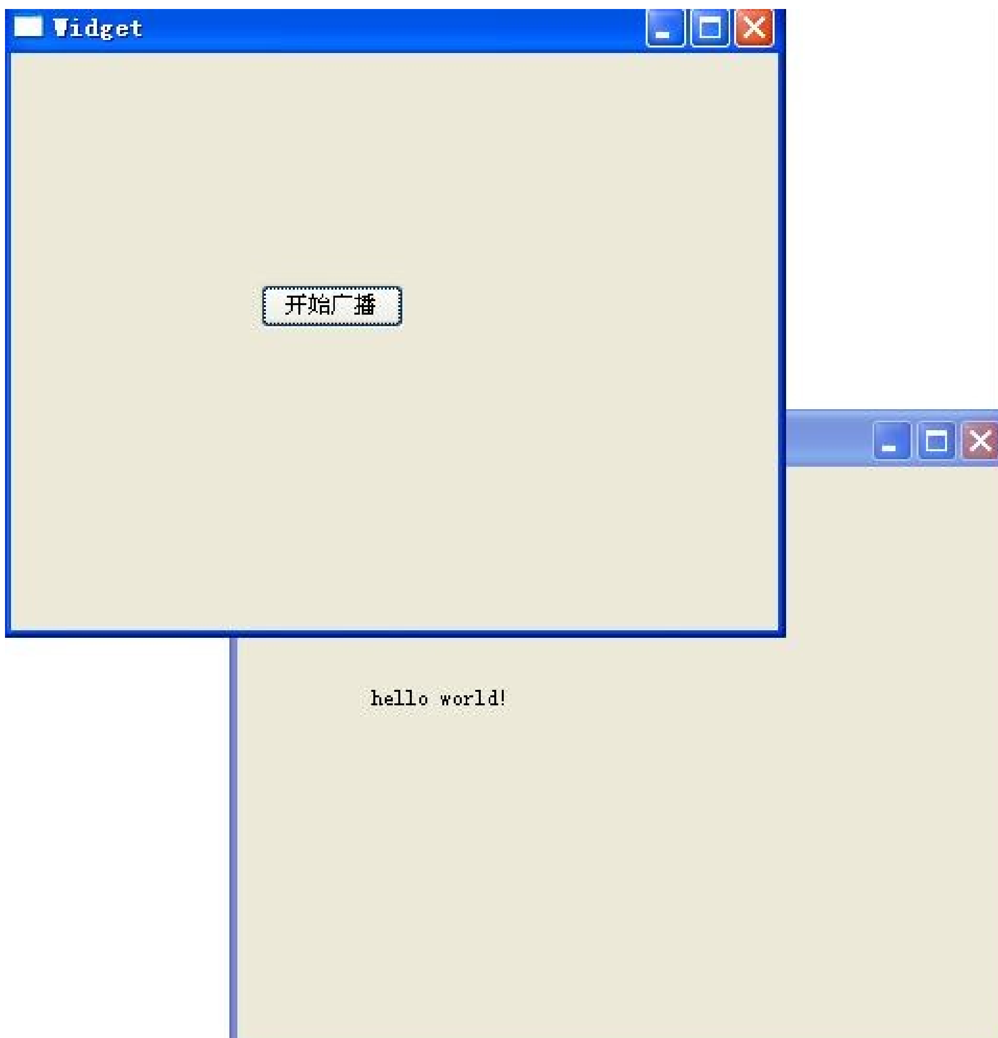
```
void Widget::processPendingDatagram() //处理等待的数据报
{
    while(receiver->hasPendingDatagrams())  //拥有等待的数据报
    {
       QByteArray datagram; //拥于存放接收的数据报
//让datagram的大小为等待处理的数据报的大小，这样才能接收到完整的数据
       datagram.resize(receiver->pendingDatagramSize());
       //接收数据报，将其存放到datagram中
       receiver->readDatagram(datagram.data(),datagram.size());
       //将数据报内容显示出来
       ui->label->setText(datagram);
    }
}
```

5、我们在项目列表中 `udpReceiver` 项目上点击鼠标右键，在弹出的菜单上选择run菜单来运行该工程。如下图所示。



6·第一次运行该程序时，系统可能会提示警告，我们选择“解除阻止”。注意，如果是在linux下，你可能还需要关闭防火墙。

7·我们同时再运行 `udpSender` 程序。然后点击其上的“发送广播”按钮，这时会在 `udpReceiver` 上显示数据报的内容。效果如下。



结语

可以看到，UDP的应用是很简单的。我们只需要在发送端执行 `writeDatagram()` 函数进行数据报的发送，然后在接收端绑定端口，并关联 `readyRead()` 信号和数据报处理函数即可。

下一节我们讲述TCP的应用。

涉及到的源码:

- udpSender.rar
- udpReceiver.rar

第37篇 网络（七）TCP（一）

导语

TCP即TransmissionControl Protocol，传输控制协议。与UDP不同，它是面向连接和数据流的可靠传输协议。也就是说，它能使一台计算机上的数据无差错的发往网络上的其他计算机，所以当要传输大量数据时，我们选用TCP协议。

TCP协议的程序使用的是客户端/服务器（C/S）模式，在Qt中提供了`QTcpSocket`类来编写客户端程序，使用`QTcpServer`类编写服务器端程序。我们在服务器端进行端口的，一旦发现客户端的连接请求，就会发出`newConnection()`信号，可以关联这个信号到我们自己的槽进行数据的发送。而在客户端，一旦有数据到来就会发出`readyRead()`信号，可以关联此信号进行数据的接收。其实，在程序中最难理解的地方就是程序的发送和接收了，为了让大家更好的理解，我们在这一节只是讲述一个传输简单的字符串的例子，在下一节再进行扩展，实现任意文件的传输。

环境：Windows Xp + Qt 4.8.5+Qt Creator2.8.0

目录

正文

一、服务器端

在服务器端的程序中，我们本地主机的一个端口，这里使用6666，然后关联`newConnection()`信号与自己写的`sendMessage()`槽。就是说一旦有客户端的连接请求，就会执行`sendMessage()`函数，在这个函数里我们发送一个简单的字符串。

1．新建QtGui应用

项目名为`tcpServer`，基类选择`QWidget`，类名为`Widget`。完成后打开项目文件`tcpServer.pro`并添加一行代码：`QT += network`，然后保存该文件。

2．在`widget.ui`的设计区添加一个Label，更改其显示文本为“等待连接”，然后更改其`objectName`为`statusLabel`，用于显示一些状态信息。

3．在`widget.h`文件中做以下更改。

添加头文件：`#include <QtNetWork>`

添加private对象：`QTcpServer *tcpServer;`

添加私有槽：

```
private slots:
void sendMessage();
```

4．在 `widget.cpp` 文件中进行更改。

在其构造函数中添加代码：

```
tcpServer = new QTcpServer(this);
if(!tcpServer->listen(QHostAddress::LocalHost,6666))
{  //本地主机的6666端口，如果出错就输出错误信息，并关闭
    qDebug() << tcpServer->errorString();
    close();
}
//连接信号和相应槽函数
connect(tcpServer,SIGNAL(newConnection()),this,SLOT(sendMessage()));
```

我们在构造函数中使用 `tcpServer` 的 `listen()` 函数进行，然后关联了 `newConnection()` 和我们自己的 `sendMessage()` 函数。

下面我们实现 `sendMessage()` 函数。

```
void Widget::sendMessage()
{
    //用于暂存我们要发送的数据
    QByteArray block;

    //使用数据流写入数据
    QDataStream out(&block,QIODevice::WriteOnly);

    //设置数据流的版本，客户端和服务器端使用的版本要相同
    out.setVersion(QDataStream::Qt_4_6);

    out<<(quint16) 0;
    out<<tr("hello Tcp!!!");
    out.device()->seek(0);
    out<<(quint16) (block.size() - sizeof(quint16));

    //我们获取已经建立的连接的子套接字
    QTcpSocket *clientConnection = tcpServer->nextPendingConnection();

    connect(clientConnection,SIGNAL(disconnected()),clientConnection,
           SLOT(deleteLater()));
    clientConnection->write(block);
    clientConnection->disconnectFromHost();

    //发送数据成功后，显示提示
    ui->statusLabel->setText("send message successful!!!");
}
```

这个是数据发送函数，我们主要介绍两点：

（1）为了保证在客户端能接收到完整的文件，我们都在数据流的最开始写入完整文件的大小信息，这样客户端就可以根据大小信息来判断是否接受到了完整的文件。而在服务器端，在发送数据时就要首先发送实际文件的大小信息，但是，文件的大小一开始是无法预知的，所以这里先使用了`out<<(quint16) 0;`在`block`的开始添加了一个`quint16`大小的空间，也就是两字节的空间，它用于后面放置文件的大小信息。然后`out<<tr("hello Tcp!!!");`输入实际的文件，这里是字符串。当文件输入完成后我们再使用`out.device()->seek(0);`返回到`block`的开始，加入实际的文件大小信息，也就是后面的代码，它是实际文件的大小：`out<<(quint16) (block.size() - sizeof(quint16));`

（2）在服务器端我们可以使用`tcpServer`的`nextPendingConnection()`函数来获取已经建立的连接的Tcp套接字，使用它来完成数据的发送和其它操作。比如这里，我们关联了`disconnected()`信号和`deleteLater()`槽，然后我们发送数据

```
clientConnection->write(block);
```

然后是`clientConnection->disconnectFromHost();`

它表示当发送完成时就会断开连接，这时就会发出`disconnected()`信号，而最后调用`deleteLater()`函数保证在关闭连接后删除该套接字`clientConnection`。

5．这样服务器的程序就完成了，可以先运行一下程序。

二、客户端

我们在客户端程序中向服务器发送连接请求，当连接成功时接收服务器发送的数据。

1．新建Qt Gui应用，

项目名`tcpClient`，基类选择`QWidget`，类名为`Widget`。完成后打开项目文件`tcpClient.pro`并添加一行代码：`QT += network`，然后保存该文件。

2．我们在`widget.ui`中添加几个标签`Label`和两个`Line Edit`以及一个按钮`Push Button`。设计效果如下图所示。



其中“主机”后的 `LineEdit` 的 `objectName` 为 `hostLineEdit` ，“端口号”后的为 `portLineEdit` 。

“收到的信息”标签的 `objectName` 为 `messageLabel` 。

3．在 `widget.h` 文件中做更改。

添加头文件： `#include <QtNetwork>`

添加 `private` 变量：

```
QTcpSocket *tcpSocket;
QString message;  //存放从服务器接收到的字符串
quint16blockSize; //存放文件的大小信息
```

添加私有槽：

```
private slots:
    void newConnect(); //连接服务器
    void readMessage();  //接收数据
void displayError(QAbstractSocket::SocketError);  //显示错误
```

4．在 `widget.cpp` 文件中做更改。

（1）在构造函数中添加代码：

```
tcpSocket = new QTcpSocket(this);
connect(tcpSocket,SIGNAL(readyRead()),this,SLOT(readMessage()));
connect(tcpSocket,SIGNAL(error(QAbstractSocket::SocketError)),
         this,SLOT(displayError(QAbstractSocket::SocketError)));
```

这里关联了 `tcpSocket` 的两个信号，当有数据到来时发出 `readyRead()` 信号，我们执行读取数据的 `readMessage()` 函数。当出现错误时发出 `error()` 信号，我们执行 `displayError()` 槽函数。

(2)实现 `newConnect()` 函数。

```
void Widget::newConnect()
{
    blockSize = 0; //初始化其为0
tcpSocket->abort(); //取消已有的连接

//连接到主机，这里从界面获取主机地址和端口号
    tcpSocket->connectToHost(ui->hostLineEdit->text(),
                             ui->portLineEdit->text().toInt());
}
```

这个函数实现了连接到服务器，下面会在"连接"按钮的单击事件槽函数中调用这个函数。

（3）实现 `readMessage()` 函数。

```
void Widget::readMessage()
{
    QDataStream in(tcpSocket);
    in.setVersion(QDataStream::Qt_4_6);
    //设置数据流版本，这里要和服务器端相同
    if(blockSize==0) //如果是刚开始接收数据
    {
       //判断接收的数据是否有两字节，也就是文件的大小信息
       //如果有则保存到blockSize变量中，没有则返回，继续接收数据
       if(tcpSocket->bytesAvailable() < (int)sizeof(quint16)) return;
       in >> blockSize;
    }
    if(tcpSocket->bytesAvailable() < blockSize) return;
    //如果没有得到全部的数据，则返回，继续接收数据
    in >> message;
    //将接收到的数据存放到变量中
    ui->messageLabel->setText(message);
    //显示接收到的数据
}
```

这个函数实现了数据的接收，它与服务器端的发送函数相对应。首先我们要获取文件的大小信息，然后根据文件的大小来判断是否接收到了完整的文件。

(4)实现 `displayError()` 函数。

```
void Widget::displayError(QAbstractSocket::SocketError)
{
    qDebug() << tcpSocket->errorString(); //输出错误信息
}
```

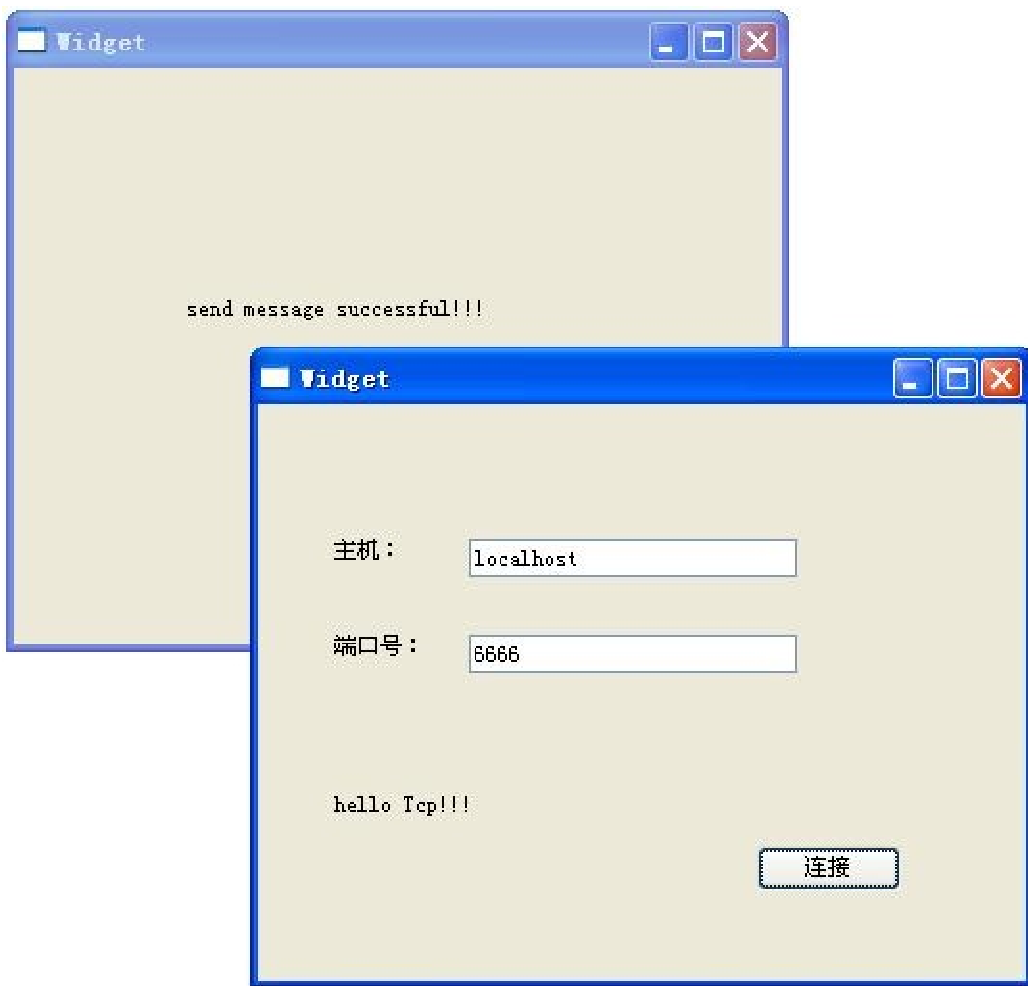
这里简单的实现了错误信息的输出。

（5）我们在 `widget.ui` 中进入"连接"按钮的单击事件槽函数，然后更改如下。

```
void Widget::on_pushButton_clicked() //连接按钮
{
    newConnect(); //请求连接
}
```

这里直接调用了 `newConnect()` 函数。

5．我们运行程序，同时运行服务器程序，然后在“主机”后填入“localhost”，在“端口号”后填入“6666”，点击“连接”按钮，效果如下。



可以看到我们正确地接收到了数据。因为服务器端和客户端是在同一台机子上运行的，所以我这里填写了“主机”为“localhost”，如果你在不同的机子上运行，需要在“主机”后填写其正确的IP地址。

结语

到这里我们最简单的TCP应用程序就完成了，在下一节我们将会对它进行扩展，实现任意文件的传输。

涉及到的源码:

- tcpServer.rar
- tcpClient.rar

第38篇 网络（八）TCP（二）

导语

在上一节里我们使用TCP服务器发送一个字符串，然后在TCP客户端进行接收。在这一节将重新写一个客户端程序和一个服务器程序，这次实现客户端进行文件的发送，服务器进行文件的接收。有了上一节的基础，这一节的内容就很好理解了，注意一下几个信号和槽的关联即可。当然，我们这次要更深入了解一下数据的发送和接收的处理方法。

环境：Windows Xp + Qt 4.8.5+QtCreator 2.8.0

目录

正文

一、客户端

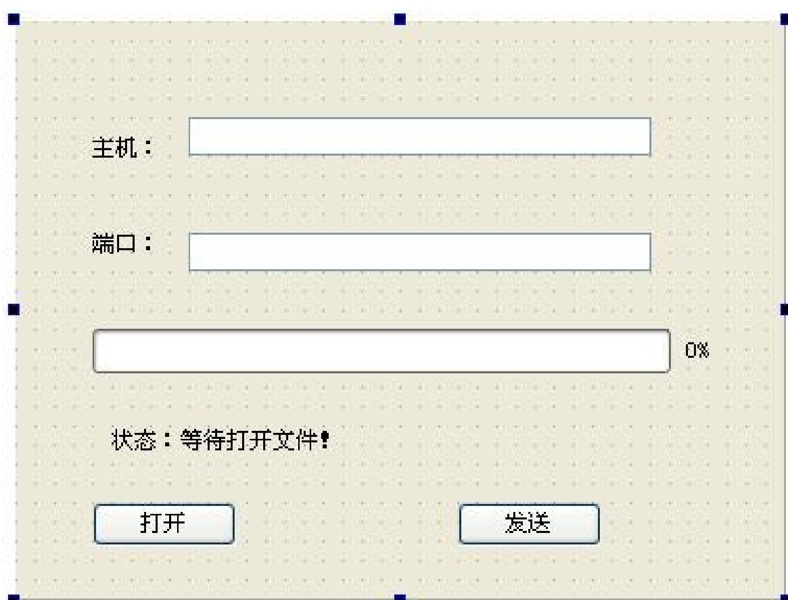
这次先讲解客户端，在客户端里需要与服务器进行连接，一旦连接成功，就会发出`connected()`信号，这时我们就进行文件的发送。

在上一节已经看到，发送数据时先发送了数据的大小信息。这一次，我们要先发送文件的总大小，然后文件名长度，然后是文件名，这三部分合称为文件头结构，最后再发送文件数据。所以在发送函数里就要进行相应的处理，当然，在服务器的接收函数里也要进行相应的处理。对于文件大小，这次使用了`qint64`，它是64位的，可以表示一个很大的文件了。

1．新建QtGui项目

名称为`tcpSender`，基类选择`QWidget`，类名为`Widget`，完成后打开`tcpSender.pro`添加一行代码：`QT += network`。

2．我们在`widget.ui`文件中将界面设计如下。



这里“主机”后的 `Line Edit` 的 `objectName` 为 `hostLineEdit` ；“端口”后的 `Line Edit` 的 `objectName` 为 `portLineEdit` ；下面的 `Progress Bar` 的 `objectName` 为 `clientProgressBar` ，其 `value` 属性设为0；“状态” `Label` 的 `objetName` 为 `clientStatusLabel` ；“打开”按钮的 `objectName` 为 `openButton` ；“发送”按钮的 `objectName` 为 `sendButton` 。

3．在 `widget.h` 文件中进行更改。

（1）添加头文件包含 `#include <QtNetwork>`

（2）添加 `private` 变量：

```
QTcpSocket *tcpClient;
    QFile *localFile;  //要发送的文件
    qint64 totalBytes;  //数据总大小
    qint64 bytesWritten;  //已经发送数据大小
    qint64 bytesToWrite;   //剩余数据大小
    qint64 loadSize;   //每次发送数据的大小
    QString fileName;  //保存文件路径
QByteArray outBlock;  //数据缓冲区，即存放每次要发送的数据
```

（3）添加私有槽函数：

```
private slots:
    void send();  //连接服务器
    void startTransfer();  //发送文件大小等信息
    void updateClientProgress(qint64); //发送数据，更新进度条
    void displayError(QAbstractSocket::SocketError); //显示错误
void openFile();  //打开文件
```

4．在 `widget.cpp` 文件中进行更改

添加头文件： `#include <QFileDialog>`

（1）在构造函数中添加代码：

```
loadSize = 4*1024;
totalBytes = 0;
bytesWritten = 0;
bytesToWrite = 0;
tcpClient = new QTcpSocket(this);
//当连接服务器成功时，发出connected()信号，我们开始传送文件
connect(tcpClient,SIGNAL(connected()),this,SLOT(startTransfer()));
//当有数据发送成功时，我们更新进度条
connect(tcpClient,SIGNAL(bytesWritten(qint64)),this,
       SLOT(updateClientProgress(qint64)));
connect(tcpClient,SIGNAL(error(QAbstractSocket::SocketError)),this,
       SLOT(displayError(QAbstractSocket::SocketError)));
//开始使"发送"按钮不可用
ui->sendButton->setEnabled(false);
```

我们主要是进行了变量的初始化和几个信号和槽函数的关联。

（2）实现打开文件函数。

```
void Widget::openFile()   //打开文件
{
    fileName = QFileDialog::getOpenFileName(this);
    if(!fileName.isEmpty())
    {
       ui->sendButton->setEnabled(true);
       ui->clientStatusLabel->setText(tr("打开文件 %1 成功！")
                                        .arg(fileName));
    }
}
```

该函数将在下面的“打开”按钮单击事件槽函数中调用。

（3）实现连接函数。

```
void Widget::send()   //连接到服务器，执行发送
{
    ui->sendButton->setEnabled(false);
    bytesWritten = 0;
    //初始化已发送字节为0
    ui->clientStatusLabel->setText(tr("连接中..."));
    tcpClient->connectToHost(ui->hostLineEdit->text(),
                             ui->portLineEdit->text().toInt());//连接
}
```

该函数将在“发送”按钮的单击事件槽函数中调用。

（4）实现文件头结构的发送。

```
void Widget::startTransfer()  //实现文件大小等信息的发送
{
    localFile = new QFile(fileName);
    if(!localFile->open(QFile::ReadOnly))
    {
       qDebug() << "open file error!";
       return;
    }

    //文件总大小
    totalBytes = localFile->size();

    QDataStream sendOut(&outBlock,QIODevice::WriteOnly);
    sendOut.setVersion(QDataStream::Qt_4_6);
QString currentFileName = fileName.right(fileName.size()
- fileName.lastIndexOf('/')-1);

    //依次写入总大小信息空间，文件名大小信息空间，文件名
    sendOut << qint64(0) << qint64(0) << currentFileName;

    //这里的总大小是文件名大小等信息和实际文件大小的总和
    totalBytes += outBlock.size();

    sendOut.device()->seek(0);
    //返回outBolock的开始，用实际的大小信息代替两个qint64(0)空间
    sendOut<<totalBytes<<qint64((outBlock.size() - sizeof(qint64)*2));

    //发送完头数据后剩余数据的大小
    bytesToWrite = totalBytes - tcpClient->write(outBlock);

    ui->clientStatusLabel->setText(tr("已连接"));
    outBlock.resize(0);
}
```

（5）下面是更新进度条，也就是发送文件数据。

```
//更新进度条，实现文件的传送
void Widget::updateClientProgress(qint64 numBytes)
{
    //已经发送数据的大小
    bytesWritten += (int)numBytes;

    if(bytesToWrite > 0) //如果已经发送了数据
    {
   //每次发送loadSize大小的数据，这里设置为4KB，如果剩余的数据不足4KB，
   //就发送剩余数据的大小
       outBlock = localFile->read(qMin(bytesToWrite,loadSize));

       //发送完一次数据后还剩余数据的大小
       bytesToWrite -= (int)tcpClient->write(outBlock);

       //清空发送缓冲区
       outBlock.resize(0);

    } else {
       localFile->close(); //如果没有发送任何数据，则关闭文件
    }

    //更新进度条
    ui->clientProgressBar->setMaximum(totalBytes);
    ui->clientProgressBar->setValue(bytesWritten);

    if(bytesWritten == totalBytes) //发送完毕
    {
     ui->clientStatusLabel->setText(tr("传送文件 %1 成功")
.arg(fileName));
       localFile->close();
       tcpClient->close();
    }
}
```

（6）实现错误处理函数。

```
void Widget::displayError(QAbstractSocket::SocketError) //显示错误
{
    qDebug() << tcpClient->errorString();
    tcpClient->close();
    ui->clientProgressBar->reset();
    ui->clientStatusLabel->setText(tr("客户端就绪"));
    ui->sendButton->setEnabled(true);
}
```

（7）我们从 `widget.ui` 中分别进行“打开”按钮和“发送”按钮的单击事件槽函数，然后更改如下。

```
void Widget::on_openButton_clicked() //打开按钮
{
    openFile();
}
void Widget::on_sendButton_clicked() //发送按钮
{
    send();
}
```

5．我们为了使程序中的中文不显示乱码，在 `main.cpp` 文件中更改。

添加头文件：`#include <QTextCodec>`

在main函数中添加代码：`QTextCodec::setCodecForTr(QTextCodec::codecForName("UTF-8"));`

6．现在可以先运行程序。

7．程序整体思路分析。

我们设计好界面，然后按下“打开”按钮，选择要发送的文件，这时调用了`openFile()`函数。然后点击“发送”按钮，调用`send()`函数，与服务器进行连接。当连接成功时就会发出`connected()`信号，这时就会执行`startTransfer()`函数，进行文件头结构的发送，当发送成功时就会发出`bytesWritten(qint64)`信号，这时执行`updateClientProgress(qint64 numBytes)`进行文件数据的传输和进度条的更新。这里使用了一个`loadSize`变量，我们在构造函数中将其初始化为`4*1024`即4字节，它的作用是，我们将整个大的文件分成很多小的部分进行发送，每部分为4字节。而当连接出现问题时就会发出`error(QAbstractSocket::SocketError)`信号，这时就会执行`displayError()`函数。对于程序中其他细节我们就不再分析，希望大家能自己编程研究一下。

二、服务器端

我们在服务器端进行数据的接收。服务器端程序是很简单的，我们开始进行监听，一旦发现有连接请求就发出`newConnection()`信号，然后我们便接受连接，开始接收数据。

1．新建QtGui应用

名称为`tcpReceiver`，基类选择`QWidget`，类名为`Widget`，完成后打开`tcpReceiver.pro`添加一行代码：`QT += network`。

2．我们更改`widget.ui`文件，设计界面如下。

其中“服务器端”`Label`的`objectName`为`serverStatusLabel`；进度条`ProgressBar`的`objectName`为`serverProgressBar`，设置其`value`属性为0；“开始监听”按钮的`objectName`为`startButton`。

效果如下。



3、更改 `widget.h` 文件的内容。

（1）添加头文件包含： `#include <QtNetwork>`

（2）添加私有变量：

```
QTcpServer tcpServer;
QTcpSocket *tcpServerConnection;
qint64 totalBytes;  //存放总大小信息
qint64 bytesReceived;  //已收到数据的大小
qint64 fileNameSize;  //文件名的大小信息
QString fileName;   //存放文件名
QFile *localFile;   //本地文件
QByteArray inBlock;   //数据缓冲区
```

（3）添加私有槽函数：

```
private slots:
    void on_startButton_clicked();
    void start();   //开始监听
    void acceptConnection();  //建立连接
void updateServerProgress();  //更新进度条，接收数据

//显示错误
void displayError(QAbstractSocket::SocketError socketError);
```

4、更改 `widget.cpp` 文件。

（1）在构造函数中添加代码：

```
totalBytes = 0;
    bytesReceived = 0;
fileNameSize = 0;

//当发现新连接时发出newConnection()信号
    connect(&tcpServer,SIGNAL(newConnection()),this,
SLOT(acceptConnection()));
```

（2）实现 `start()` 函数。

```
void Widget::start() //开始监听
{
    ui->startButton->setEnabled(false);
    bytesReceived =0;
    if(!tcpServer.listen(QHostAddress::LocalHost,6666))
    {
       qDebug() << tcpServer.errorString();
       close();
       return;
    }
    ui->serverStatusLabel->setText(tr("监听"));
}
```

（3）实现接受连接函数。

```
void Widget::acceptConnection()  //接受连接
{
    tcpServerConnection = tcpServer.nextPendingConnection();
connect(tcpServerConnection,SIGNAL(readyRead()),this,
SLOT(updateServerProgress()));
    connect(tcpServerConnection,
SIGNAL(error(QAbstractSocket::SocketError)),this,
           SLOT(displayError(QAbstractSocket::SocketError)));
    ui->serverStatusLabel->setText(tr("接受连接"));
    tcpServer.close();
}
```

（4）实现更新进度条函数。

```
void Widget::updateServerProgress()  //更新进度条，接收数据
{
   QDataStream in(tcpServerConnection);
   in.setVersion(QDataStream::Qt_4_6);
   if(bytesReceived <= sizeof(qint64)*2)
   { //如果接收到的数据小于16个字节，那么是刚开始接收数据，我们保存到//来的头文件信息
       if((tcpServerConnection->bytesAvailable() >= sizeof(qint64)*2)
           && (fileNameSize == 0))
       { //接收数据总大小信息和文件名大小信息
           in >> totalBytes >> fileNameSize;
           bytesReceived += sizeof(qint64) * 2;
       }
       if((tcpServerConnection->bytesAvailable() >= fileNameSize)
           && (fileNameSize != 0))
       {  //接收文件名，并建立文件
           in >> fileName;
           ui->serverStatusLabel->setText(tr("接收文件 %1 ...")
                                           .arg(fileName));
           bytesReceived += fileNameSize;
           localFile= new QFile(fileName);
           if(!localFile->open(QFile::WriteOnly))
           {
                qDebug() << "open file error!";
                return;
           }
       }
       else return;
   }
   if(bytesReceived < totalBytes)
   {  //如果接收的数据小于总数据，那么写入文件
      bytesReceived += tcpServerConnection->bytesAvailable();
      inBlock= tcpServerConnection->readAll();
      localFile->write(inBlock);
      inBlock.resize(0);
   }
//更新进度条
   ui->serverProgressBar->setMaximum(totalBytes);
   ui->serverProgressBar->setValue(bytesReceived);

   if(bytesReceived == totalBytes)
   { //接收数据完成时
    tcpServerConnection->close();
    localFile->close();
    ui->startButton->setEnabled(true);
ui->serverStatusLabel->setText(tr("接收文件 %1 成功！")
.arg(fileName));
   }
}
```

（5）错误处理函数。

```
void Widget::displayError(QAbstractSocket::SocketError) //错误处理
{
    qDebug() << tcpServerConnection->errorString();
    tcpServerConnection->close();
    ui->serverProgressBar->reset();
    ui->serverStatusLabel->setText(tr("服务端就绪"));
    ui->startButton->setEnabled(true);
}
```

（6）我们在 `widget.ui` 中进入“开始监听”按钮的单击事件槽函数，更改如下。

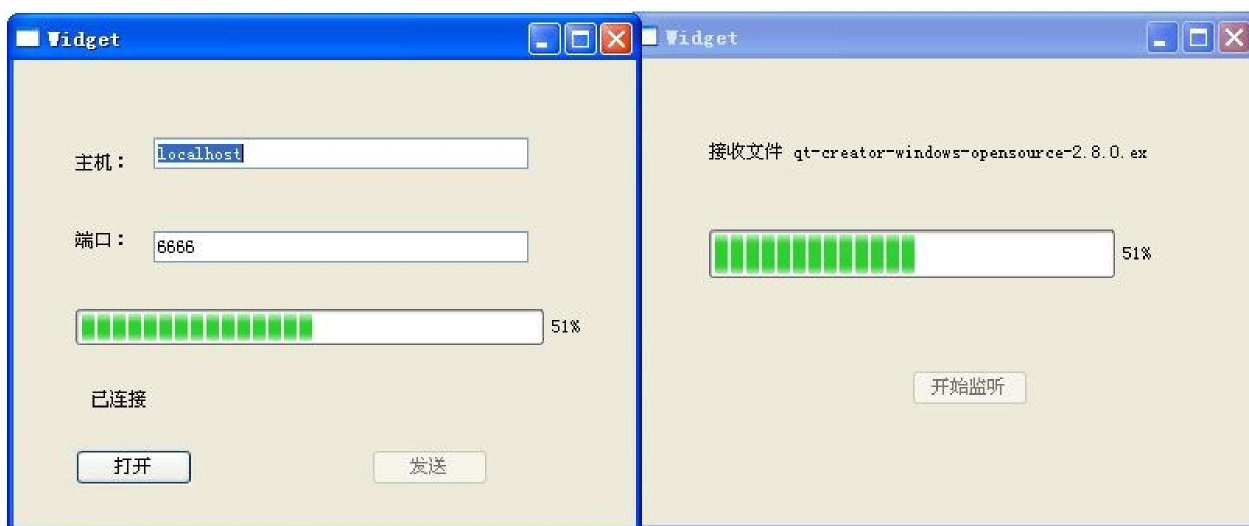
```
void Widget::on_startButton_clicked() //开始监听按钮
{
    start();
}
```

5．我们为了使程序中的中文不显示乱码，在 `main.cpp` 文件中更改。

添加头文件包含：`#include <QTextCodec>`

在 `main` 函数中添加代码：`QTextCodec::setCodecForTr(QTextCodec::codecForName("UTF-8"));`

6．运行程序，并同时运行 `tcpSender` 程序，效果如下。



我们先在服务器端按下“开始监听”按钮，然后在客户端输入主机地址和端口号，然后打开要发送的文件，点击“发送”按钮进行发送。

结语

在这两节里我们介绍了TCP的应用，可以看到服务器端和客户度端都可以当做发送端或者接收端，而且数据的发送与接收只要使用相对应的协议即可，它是可以根据用户的需要来进行编程的，没有固定的格式。《Qt及Qt Quick开发实战精解》中的局域网聊天工具就是本节知识的扩展，大家可以从社区下载页面下载其源码。

涉及到的源码:

- tcpSender.rar
- tcpReceiver.rar

第39篇 网络（九）进程和线程

导语

在前面的几节内容中讲解了Qt网络编程的一些基本内容，这一节来看一下在Qt中进程和线程的基本应用。

环境：Windows Xp + Qt 4.8.5+Qt Creator2.8.0

目录

- 一、进程
- 二、线程

正文

一、进程

在设计一个应用程序时，有时不希望将一个不太相关的功能集成到程序中，或者是因为该功能与当前设计的应用程序联系不大，或者是因为该功能已经可以使用现成的程序很好的实现了，这时就可以在当前的应用程序中调用外部的程序来实现该功能，这就会使用到进程。Qt应用程序可以很容易的启动一个外部应用程序，而且Qt也提供了在多种进程间通信的方法。

Qt的 `QProcess` 类用来启动一个外部程序并与其进行通信。下面我们来看一下怎么在Qt代码中启动一个进程。

1．首先创建QtGui应用。

工程名称为 `myProcess` ，其他选项保持默认即可。

2．然后设计界面。

在设计模式往界面上拖入一个 `Push Button` 部件，修改其显示文本为“启动一个进程”。

3．修改槽。

在按钮上点击鼠标右键，转到其 `clicked()` 信号对应的槽，更改如下：

```
void MainWindow::on_pushButton_clicked()
{
     myProcess.start("notepad.exe");
}
```

4．进入`mainwindow.h`文件添加代码。

先添加头文件包含：`#include <QProcess>`，然后添加私有对象定义：`QProcess myProcess;`

5．运行程序。

当单击界面上的按钮时就会弹出一个记事本程序。

这里我们使用`QProcess`对象运行了Windows系统下的记事本程序（即`notepad.exe`程序），因为该程序在系统目录中，所以这里不需要指定其路径。大家也可以运行其他任何的程序，只需要指定其具体路径即可。我们看到，可以使用`start()`来启动一个程序，有时启动一个程序时需要指定启动参数，这种情况在命令行启动程序时是很常见的，下面来看一个例子，还在前面的例子的基础上进行更改。

1．在`mainwindow.h`文件中添加代码。

添加私有槽：

```
private slots:
    void showResult();
```

2．在`mainwindow.cpp`文件中添加代码。

（1）先添加头文件包含：`#include <QDebug>`，然后在构造函数中添加如下代码：

```
connect(&myProcess,SIGNAL(readyRead()), this, SLOT(showResult()));
```

（2）然后添加`showResult()`槽的定义：

```
void MainWindow::showResult()
{
    qDebug() << "showResult: " << endl
            << QString(myProcess.readAll());
}
```

（3）最后将前面按钮的单击信号对应的槽更改为：

```
void MainWindow::on_pushButton_clicked()
{
    QString program = "cmd.exe";
    QStringList arguments;
    arguments << "/c dir&pause";
    myProcess.start(program, arguments);
}
```

这里在启动Windows下的命令行提示符程序`cmd.exe`时为其提供了命令作为参数，这样可以直接执行该命令。当命令执行完以后可以执行`showResult()`槽来显示运行的结果。这里为了可以显示结果中的中文字符，使用了`QString()`进行编码转换。这需要在`mian()`函数中添加代码。

3 · 为了确保可以显示输出的中文字符，在 `main.cpp` 文件中添加代码。 先添加头文件包含 `#include <QTextCodec>` ，然后在 `main()` 函数第一行代码下面，添加如下一行代码：

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
```

4 · 运行程序。

按下界面上的按钮，会在Qt Creator中的应用程序输出栏中输出命令的执行结果。

对于Qt中进程进一步的使用可以参考 `QProcess` 类的帮助文档。在Qt中还提供了多种进程间通信的方法，大家可以在Qt帮助中查看Inter-ProcessCommunication in Qt关键字对应的文档。

二、线程

Qt提供了对线程的支持，这包括一组与平台无关的线程类，一个线程安全的发送事件的方式，以及跨线程的信号-槽的关联。这些使得可以很容易的开发可移植的多线程Qt应用程序，可以充分利用多处理器的机器。多线程编程也可以有效的解决在不冻结一个应用程序的用户界面的情况下执行一个耗时的操作的问题。关于线程的内容，大家可以在Qt帮助中参考Thread Support in Qt关键字。

（一）启动一个线程

Qt中的 `QThread` 类提供了平台无关的线程。一个 `QThread` 代表了一个在应用程序中可以独立控制的线程，它与进程中的其他线程分享数据，但是是独立执行的。相对于一般的程序都是从 `main()` 函数开始执行， `QThread` 从 `run()` 函数开始执行。默认的， `run()` 通过调用 `exec()` 来开启事件循环。要创建一个线程，需要子类化 `QThread` 并且重新实现 `run()` 函数。

每一个线程可以有自己的事件循环，可以通过调用 `exec()` 函数来启动事件循环，可以通过调用 `exit()` 或者 `quit()` 来停止事件循环。在一个线程中拥有一个事件循环，可以使它能够关联其他线程中的信号到本线程的槽上，这使用了队列关联机制，就是在使用 `connect()` 函数进行信号和槽的关联时，将 `Qt::ConnectionType` 类型的参数指定为 `Qt::QueuedConnection` 。拥有事件循环还可以使该线程能过使用需要事件循环的类，比如 `QTimer` 和 `QTcpSocket` 类等。注意，在线程中是无法使用任何的部件类的。

下面来看一个在图形界面程序中启动一个线程的例子，在界面上有两个按钮，一个用于开启一个线程，一个用于关闭该线程。

1 · 创建项目。

新建Qt Gui应用，名称为 `myThread` ，类名为 `Dialog` ，基类选择 `QDialog` 。

2 · 设计界面。

完成项目创建后进入设计模式，向界面中放入两个 `Push Button` 按钮，将第一个按钮的显示文本更改为“启动线程”，将其 `objectName` 属性更改为 `startButton` ；将第二个按钮的显示文本更改为“终止线程”，将其 `objectName` 属性更改为 `stopButton` ，将其 `enabled` 属性取消选中。

3. 添加自定义线程类。

向项目中添加新的C++类，类名设置为 `MyThread` ，基类设置为 `QThread` ，类型信息选择“继承自QObject”。完成后进入 `mythread.h` 文件，先添加一个公有函数声明：

```
void stop();
```

然后再添加一个函数声明和一个变量的定义：

```
protected:
    void run();
private:
    volatile bool stopped;
```

这里 `stopped` 变量使用了 `volatile` 关键字，这样可以使它在任何时候都保持最新的值，从而可以避免在多个线程中访问它时出错。然后进入 `mythread.cpp` 文件中，先添加头文件 `#include <QDebug>` ，然后在构造函数中添加如下代码：

```
stopped = false;
```

这里将 `stopped` 变量初始化为 `false` 。下面添加 `run()` 函数的定义：

```
void MyThread::run()
{
    qreal i = 0;
    while (!stopped)
        qDebug() << QString("in MyThread: %1").arg(i++);
    stopped = false;
}
```

这里一直判断 `stopped` 变量的值，只要它为 `false` ，那么就一直打印字符串。下面添加 `stop()` 函数的定义：

```
void MyThread::stop()
{
    stopped = true;
}
```

在 `stop()` 函数中将 `stopped` 变量设置为了 `true` ，这样便可以结束 `run()` 函数中的循环，从而从 `run()` 函数中退出，这样整个线程也就结束了。这里使用了 `stopped` 变量来实现了进程的终止，并没有使用危险的 `terminate()` 函数。

4. 在 `Dialog` 类中使用自定义的线程。

先到 `dialog.h` 文件中，添加头文件包含：

```
#include "mythread.h"
```

然后添加私有对象的定义：

```
MyThread thread;
```

下面到设计模式，分别进入两个按钮的单击信号对应的槽，更改如下：

```
// 启动线程按钮
void Dialog::on_startButton_clicked()
{
    thread.start();
    ui->startButton->setEnabled(false);
    ui->stopButton->setEnabled(true);
}

// 终止线程按钮
void Dialog::on_stopButton_clicked()
{
    if (thread.isRunning()) {
        thread.stop();
        ui->startButton->setEnabled(true);
        ui->stopButton->setEnabled(false);
    }
}
```

在启动线程时调用了 `start()` 函数，然后设置了两个按钮的状态。在终止线程时，先使用 `isRunning()` 来判断线程是否在运行，如果是，则调用 `stop()` 函数来终止线程，并且更改两个按钮的状态。现在运行程序，按下“启动线程”按钮，查看应用程序输出栏的输出，然后再按下“终止线程”按钮，可以看到已经停止输出了。

下面我们接着来优化这个程序，通过信号和槽来将子线程中的字符串显示到主界面上。

1. 在 `mythread.h` 文件中添加信号的定义：

```
signals:
void stringChanged(const QString &);
```

2. 然后到 `mythread.cpp` 文件中更改 `run()` 函数的定义：

```
void MyThread::run()
{
    long int i = 0;
    while (!stopped) {
       QString str = QString("in MyThread: %1").arg(i);
       emit stringChanged(str);
       msleep(1000);
       i++;
    }
    stopped = false;
}
```

这里每隔1秒就发射一次信号，里面包含了生成的字符串。

3．到 `dialog.h` 文件中添加槽声明：

```
private slots:
    void changeString(const QString &);
```

4．打开 `dialog.ui` ，然后向主界面上拖入一个 `Label` 标签部件。

5．到 `dialog.cpp` 文件中，在构造函数里面添加信号和槽的关联：

```
// 关联线程中的信号和本类中的槽
connect(&thread, SIGNAL(stringChanged(QString)),
this, SLOT(changeString(QString)));
```

6．然后添加槽的定义：

```
void Dialog::changeString(const QString &str)
{
    ui->label->setText(str);
}
```

这里就是将子线程发送过来的字符串显示到主界面上。现在可以运行程序，查看效果了。

（二）线程同步

Qt中的 `QMutex` 、 `QReadWriteLock` 、 `QSemaphore` 和 `QWaitCondition` 类提供了同步线程的方法。虽然使用线程的思想是多个线程可以尽可能的并发执行，但是总有一些时刻，一些线程必须停止来等待其他线程。例如，如果两个线程尝试同时访问相同的全局变量，结果通常是不确定的。 `QMutex` 提供了一个互斥锁（ `mutex` ）； `QReadWriteLock` 即读-写锁； `QSemaphore` 即信号量； `QWaitCondition` 即条件变量。

（三）可重入与线程安全

在查看Qt的帮助文档时，在很多类的开始都写着“All functions in this class are reentrant”，或者“All functions in this class are thread-safe”。在Qt文档中，术语“可重入（reentrant）”和“线程安全（thread-safe）”用来标记类和函数，来表明怎样在多线程应用程序中使用它们：

一个线程安全的函数可以同时被多个线程调用，即便是这些调用使用了共享数据。因为该共享数据的所有实例都被序列化了。

一个可重入的函数也可以同时被多个线程调用，但是只能是在每个调用使用自己的数据时。

结语

最后要注意的是，使用线程是很容易出现问题的，比如无法在主线程以外的线程中使用GUI类的问题（可以简单的通过这样的方式来解决：将一些非常耗时的操作放在一个单独的工作线程中来进行，等该工作线程完成后将结果返回给主线程，最后由主线程将结果显示到屏幕上）。大家应该谨慎的使用线程。

涉及到的源码下载

第40篇 网络（十）WebKit初识

导语

WebKit是一个开源的浏览器引擎。Qt中提供了基于WebKit的QtWebKit模块，它包含了一组相关的类。QtWebKit提供了一个Web浏览器引擎，使用它便可以很容易的将万维网（WorldWide Web）中的内容嵌入到Qt应用程序中。与此同时，本地也可以对Web内容进行控制。QtWebKit可以呈现HTML（HyperTextMarkup Language，超文本标记语言）文档、XHTML（Extensible HyperTextMarkup Language，可扩展超文本标记语言）文档和SVG（Scalable VectorGraphics，可缩放矢量图形）文档，风格使用CSS（Cascading StyleSheets，层叠样式表），脚本使用JavaScript。在JavaScript执行环境和Qt对象模型间搭建的桥梁，实现了使用WebKit的JavaScript环境访问本地对象。关于这一点，大家可以在帮助中参考The QtWebKit Bridge关键字对应的文档。通过整合Qt的网络模块，实现了从Web服务器、本地文件系统甚至Qt资源系统中透明的加载Web页面。

环境：Windows Xp + Qt 4.8.5+Qt Creator2.8.0

目录

正文

一、简单应用

下面我们来实现一个可以打开特定网页的程序。新建空的Qt项目，在 `pro` 项目文件中添加一行代码： `QT += webkit` ，然后向项目中添加一个 `main.cpp` 文件，并在其中添加如下代码：

```
#include <QWebView>
#include <QApplication>
int main(int argc, char* argv[])
{
    QApplication a(argc, argv);
    QWebView view;
    view.load(QUrl("http://www.qter.org"));
    view.show();
    return a.exec();
}
```

要使用WebKit，就要先添加`webkit`模块。`QWebView`是`QtWebKit`模块主要的窗体部件，它可以在各种应用程序中用来显示Internet上的网页内容。`QWebView`作为一个窗口部件，可以嵌入到窗体或者图形视图部件中。

`QWebView`用来显示Web页面，每个`QWebView`实例都包含一个`QWebPage`对象。`QWebPage`提供了对一个页面的文档结构的访问，描述了如框架（frame）、访问历史记录和可编辑内容的撤销/重做栈等特色。每一个`QWebPage`都包含一个`QWebFrame`对象作为它的主框架。在HTML中的每一个单独的框架都可以使用`QWebFrame`来表示，这个类包含了到JavaScript窗口对象的桥梁，而且可以进行绘制。在QWebPage的主框架中可以包含很多的子框架。

HTML文档中单独的元素可以通过DOM JavaScript接口进行访问，在`QtWebKit`中与这个接口等价的接口由`QWebElement`来表示。`QWebElement`对象可以使用`QWebFram`e
的`findAllElement()`和`findFirstElement()`函数来获取。一般的网页浏览器的特色设置都可以通过`QWebSettings`类来配置，可以通过默认设置为所有的`QWebPage`实例提供默认值。单独的属性可以使用页面指定的设置对象进行重写。

二、扩展应用

下面再来看一个可以随意更改网址并且可以显示网站logo的例子。新建Qt Gui应用，项目名称为`webview`，类名和基类保持`MainWindow`和`QMainWindow`不变。完成后向`webview.pro`文件中添加`QT += webkit`一行代码，并按下`Ctrl + S`保存该文件。

1．下面到`mainwindow.h`文件中，先添加头文件：

```
#include <QWebView>
#include <QLineEdit>
```

然后添加槽的声明：

```
protected slots:
    void changeLocation();      // 改变路径
    void setProgress(int);      // 更新进度
    void adjustTitle();         // 更新标题显示
void finishLoading(bool);   // 加载完成后进行处理
       再添加对象和变量定义：
QWebView *view;
QLineEdit *locationEdit;
int progress;
```

2．下面到`mainwindow.cpp`文件中，在构造函数中添加如下代码：

```
progress = 0;
view = new QWebView(this);
setCentralWidget(view);
resize(800, 600);

// 关联信号和槽
connect(view, SIGNAL(loadProgress(int)), this, SLOT(setProgress(int)));
connect(view, SIGNAL(titleChanged(QString)), this, SLOT(adjustTitle()));
connect(view, SIGNAL(loadFinished(bool)), this, SLOT(finishLoading(bool)));
locationEdit = new QLineEdit(this);
connect(locationEdit, SIGNAL(returnPressed()), this, SLOT(changeLocation()));

// 向工具栏添加动作和部件
ui->mainToolBar->addAction(view->pageAction(QWebPage::Back));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Forward));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Reload));
ui->mainToolBar->addAction(view->pageAction(QWebPage::Stop));
ui->mainToolBar->addWidget(locationEdit);

// 设置并加载初始网页地址
locationEdit->setText("http://www.baidu.com");
view->load(QUrl("http://www.baidu.com"));
```

当 `QWebView` 开始加载时，会发射 `loadStarted()` 信号；而每当一个网页元素（例如一张图片或一个脚本等）加载完成时，都会发射 `loadProgress()` 信号；最后，当加载全部完成后，会发射 `loadFinished()` 信号，如果加载成功，该函数的参数为 `true` ，否则为 `false` 。可以使用 `title()` 来获取HTML文档的标题，如果标题发生了改变，将会发射 `titleChanged()` 信号。

3. 下面添加那几个槽的定义：

```
void MainWindow::changeLocation()
{
    QUrl url = QUrl(locationEdit->text());
    view->load(url);
    view->setFocus();
}
void MainWindow::setProgress(int p)
{
    progress = p;
    adjustTitle();
}
void MainWindow::adjustTitle()
{
    if ( progress <= 0 || progress >= 100) {
       setWindowTitle(view->title());
    } else {
setWindowTitle(QString("%1 (%2%)").arg(view->title()).arg(progress));
    }
}
void MainWindow::finishLoading(bool finished)
{
    if (finished) {
       progress = 100;
       setWindowTitle(view->title());
    } else {
       setWindowTitle("web page loading error!");
    }
}
```

下面运行程序，效果如下图所示：



结语

WebKit是一个很庞大的体系，我们这里只是讲解了其最基本的应用，有兴趣的朋友可以结合Qt文档来进一步的学习。

进阶篇

第43篇 进阶（三）对象树与拥有权

导语

学习完前面的内容，大家对应用Qt编程应该已经有了一个大概的印象。后面的内容我们将介绍Qt中的一些核心机制，它们是构成Qt的基础，包括对象模型、信号和槽、对象树与拥有权等。在前面使用Qt编程时，大家对一些内容可能存在疑惑，学习完下面的知识，可以帮助大家更好的使用Qt进行编程。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

正文

一、对象模型

标准C++对象模型可以在运行时非常有效的支持对象范式（object paradigm），但是它的静态特性在一些问题领域中不够灵活。图形用户界面编程不仅需要运行时的高效性，还需要高度的灵活性。为此，Qt在标准C++对象模型的基础上添加了一些特性，形成了自己的对象模型。这些特性有：

- 一个强大的无缝对象通信机制——信号和槽（signals and slots）；
- 可查询和可设计的对象属性系统（object properties）；
- 强大的事件和事件过滤器（events and event filters）；
- 通过上下文进行国际化的字符串翻译机制（string translation for internationalization）；
- 完善的定时器（timers）驱动，使得可以在一个事件驱动的GUI中处理多个任务；
- 分层结构的、可查询的对象树（object trees），它使用一种很自然的方式来组织对象拥有权（object ownership）；
- 守卫指针即 `QPointer` ，它在引用对象被销毁时自动将其设置为0；
- 动态的对象转换机制（dynamic cast）；

Qt的这些特性都是在遵循标准C++规范内实现的，使用这些特性都必须要继承自 `QObject` 类。其中对象通信机制和动态属性系统，还需要元对象系统（Meta-ObjectSystem）的支持。关于对象模型的介绍，大家可以在帮助中查看Object Model关键

字。

二、元对象系统

Qt中的元对象系统（Meta-Object System）提供了对象间通信的信号和槽机制、运行时类型信息和动态属性系统。元对象系统是基于以下三个条件的：

- 该类必须继承自`QObject`类；
- 必须在类的私有声明区声明`Q_OBJECT`宏（在类定义时，如果没有指定`public`或者`private`，则默认为private）；
- 元对象编译器Meta-Object Compiler（moc），为`QObject`的子类实现元对象特性提供必要的代码。

其中moc工具读取一个C++源文件，如果它发现一个或者多个类的声明中包含有`Q_OBJECT`宏，便会另外创建一个C++源文件（就是在项目目录中的`debug`目录下看到的以moc开头的C++源文件），其中包含了为每一个类生成的元对象代码。这些产生的源文件或者被包含进类的源文件中，或者和类的实现同时进行编译和链接。

元对象系统主要是为了实现信号和槽机制才被引入的，不过除了信号和槽机制以外，元对象系统还提供了其他一些特性：

- `QObject::metaObject()`函数可以返回一个类的元对象，它是`QMetaObject`类的对象；
- `QMetaObject::className()`可以在运行时以字符串形式返回类名，而不需要C++编辑器原生的运行时类型信息（RTTI）的支持；
- `QObject::inherits()`函数返回一个对象是否是`QObject`继承树上一个类的实例的信息；
- `QObject::tr()`和`QObject::trUtf8()`进行字符串翻译来实现国际化；
- `QObject::setProperty()`和`QObject::property()`通过名字来动态设置或者获取对象属性；
- `QMetaObject::newInstance()`构造该类的一个新实例。

除了这些特性外，还可以使用`qobject_cast()`函数来对`QObject`类进行动态类型转换，这个函数的功能类似于标准C++中的`dynamic_cast()`函数，但它不再需要RTTI的支持。这个函数尝试将它的参数转换为尖括号中的类型的指针，如果是正确的类型则返回一个非零的指针，如果类型不兼容则返回0。例如：

```
QObject *obj = new MyWidget;
QWidget *widget = qobject_cast<QWidget *>(obj);
```

信号和槽机制是Qt的核心内容，而信号和槽机制必须依赖于元对象系统，所以它是Qt中很关键的内容。关于元对象系统的知识，可以在Qt中查看The Meta-Object System关键字。

三、对象树与拥有权

Qt中使用对象树（object tree）来组织和管理所有的 `QObject` 类及其子类的对象。当创建一个 `QObject` 时，如果使用了其他的对象作为其父对象（parent），那么这个 `QObject` 就会被添加到父对象的 `children()` 列表中，这样当父对象被销毁时，这个 `QObject` 也会被销毁。实践表明，这个机制非常适合于管理GUI对象。例如，一个 `QShortcut` （键盘快捷键）对象是相应窗口的一个子对象，所以当用户关闭了这个窗口时，这个快捷键也可以被销毁。

`QWidget` 作为能够在屏幕上显示的所有部件的基类，扩展了对象间的父子关系。一个子对象一般也就是一个子部件，因为它们要显示在父部件的区域之中。例如，当关闭一个消息对话框（message box）后要销毁它时，消息对话框中的按钮和标签也会被销毁，这也正是我们所希望的，因为按钮和标签是消息对话框的子部件。当然，我们也可以自己来销毁一个子对象。关于这一部分的内容，大家可以在帮助索引中查看Object Trees &Ownership关键字。

在前面的Qt编程中我们应该看到过很多使用 `new` 来创建一个部件，但是却没有使用 `delete` 来进行释放的问题。这里再来研究一下这个问题。

新建Qt Gui应用，项目名称为 `myOwnership` ，基类选择 `QWidget` ，然后类名保持 `Widget` 不变。完成后向项目中添加新文件，模板选择C++ Class，类名为 `MyButton` ，基类为 `QPushButton` ，类型信息选择“继承自 `QWidget` ”。添加完文件后在 `mybutton.h` 文件中添加析构函数的声明：

```
~MyButton();
```

然后到 `mybutton.cpp` 文件中添加头文件 `#include <QDebug>` 并定义析构函数：

```
MyButton::~MyButton()
{
    qDebug() << "delete button";
}
```

这样当 `MyButton` 的对象被销毁时，就会输出相应的信息。这里定义析构函数，只是为了更清楚的看到部件的销毁过程，其实一般在构建新类时不需要实现析构函数。下面在 `widget.cpp` 文件中进行更改，添加头文件：

```
#include "mybutton.h"
#include<QDebug>
```

在构造函数中添加代码：

```
MyButton *button = new MyButton(this);    // 创建按钮部件，指定widget为父部件
button->setText(tr("button"));
```

更改析构函数：

```
Widget::~Widget()
{
    delete ui;
    qDebug() << "delete widget";
}
```

`Widget` 类的析构函数中默认的已经有了销毁 `ui` 的语句，这里又添加了输出语句。当 `Widget` 窗口被销毁时，将输出信息。下面运行程序，然后关闭窗口，在QtCreator的应用程序输出栏中的输出信息为：

```
delete widget
delete button
```

可以看到，当关闭窗口后，因为该窗口是顶层窗口，所以应用程序要销毁该窗口部件（如果不是顶层窗口，那么关闭时只是隐藏，不会被销毁），而当窗口部件销毁时会自动销毁其子部件。这也就是为什么在Qt中经常只看到 `new` 操作而看不到 `delete` 操作的原因。再来看一下 `main.cpp` 文件，其中 `Widget` 对象是建立在栈上的：

```
Widget w;
w.show();
```

这样对于对象w，在关闭程序时会被自动销毁。而对于 `Widget` 中的部件，如果是在堆上创建（使用 `new` 操作符），那么只要指定 `Widget` 为其父窗口就可以了，也不需要进行 `delete` 操作。整个应用程序关闭时，会去销毁 `w` 对象，而此时又会自动销毁它的所有子部件，这些都是`Qt的对象树所完成的。

所以，对于规范的Qt程序，我们要在 `main()` 函数中将主窗口部件创建在栈上，例如 `Widget w;` ，而不要在堆上进行创建（使用 `new` 操作符）。对于其他窗口部件，可以使用 `new` 操作符在堆上进行创建，不过一定要指定其父部件，这样就不需要再使用 `delete` 操作符来销毁该对象了。

还有一种重定义父部件（reparented）的情况，例如，将一个包含其他部件的布局管理器应用到窗口上，那么该布局管理器和其中的所有部件都会自动将它们的父部件转换为该窗口部件。在 `widget.cpp` 文件中添加头文件 `#include <QHBoxLayout>` ，然后在构造函数中继续添加代码：

```
MyButton *button2 = new MyButton;
MyButton *button3 = new MyButton;
QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(button2);
layout->addWidget(button3);
setLayout(layout);      // 在该窗口中使用布局管理器
```

这里创建了两个 `MyButton` 和一个水平布局管理器，但是并没有指定它们的父部件，现在各个部件的拥有权（ownership）不是很清楚。但是当使用布局管理器来管理这两个按钮，并且在窗口中使用这个布局管理器后，这两个按钮和水平布局管理器都将重定义父部件而成为窗

口 `Widget` 的子部件。可以使用 `children()` 函数来获取一个部件的所有子部件的列表，例如在构造函数中再添加如下代码：

```
qDebug() << children();    // 输出所有子部件的列表
```

这时大家可以运行一下程序，查看应用程序输出栏中的信息，然后根据自己的想法更改一下程序，来进一步体会Qt中对象树的概念。

结语

Qt中的对象树很好地解决了父子部件的关系，对于Gui编程是十分方便的，在创建部件时我们只需要关注它的父部件，这样就不用再考虑其销毁问题了。下一节，我们将讲解Qt中的信号和槽的内容。

涉及到的源码

第44篇 进阶（四）信号和槽

导语

在前面的内容中已经多次用到过信号和槽了，这一节我们将详细讲解信号和槽的机制和使用方式。大家可以在帮助中查看Signals& Slots关键字。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

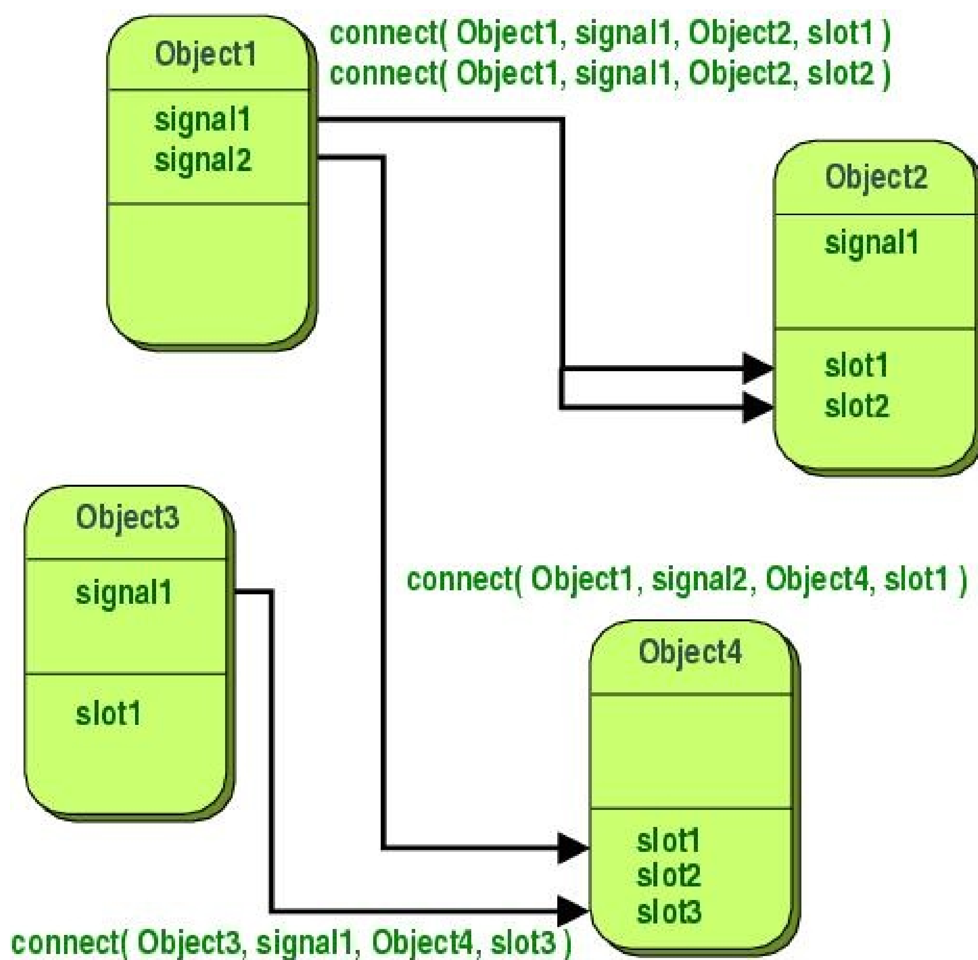
- 一、信号和槽机制
- 二、信号和槽的自动关联
- 三、信号和槽的高级应用

正文

一、信号和槽机制

信号和槽用于两个对象之间的通信，信号和槽机制是Qt的核心特征，也是Qt不同于其他开发框架的最突出的特征。在GUI编程中，当改变了一个部件时，总希望其他部件也能了解到该变化。更一般来说，我们希望任何对象都可以和其他对象进行通信。例如，如果用户点击了关闭按钮，我们希望可以执行窗口的 `close()` 函数来关闭窗口。为了实现对象间的通信，一些工具包中使用了回调（callback）机制，而在Qt中，使用了信号和槽来进行对象间的通信。当一个特殊的事情发生时便可以发射一个信号，比如按钮被单击；而槽就是一个函数，它在信号发射后被调用，来响应这个信号。在Qt的部件类中已经定义了一些信号和槽，但是更多的做法是子类化这个部件，然后添加自己的信号和槽来实现想要的功能。

在前面使用过的信号和槽的关联，都是一个信号对应一个槽。其实，一个信号可以关联到多个槽上，多个信号也可以关联到同一个槽上，甚至，一个信号还可以关联到另一个信号上，如下图所示。如果存在多个槽与某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，但是它们执行的顺序是随机的，无法指定它们的执行顺序。



下面通过一个简单的例子来进一步讲解信号和槽的相关知识。这个例子实现的效果是：在主界面中创建一个对话框，在这个对话框中可以输入数值，当按下确定按钮时关闭对话框并且将输入的数值通过信号发射出去，而在主界面中接收该信号并且显示数值。

新建Qt Gui应用，项目名称为 `mySignalSlot` ，基类选择 `QWidget` ，然后类名保持 `Widget` 不变。项目建立完成后，向项目中添加新文件，模板选择Qt分类中的“Qt设计师界面类”，界面模板选择 `Dialog without Buttons` ，类名为 `MyDialog` 。完成后首先在 `mydialog.h` 文件中添加代码来声明一个信号：

```
signals:
    void dlgReturn(int);                    // 自定义的信号
```

声明一个信号要使用 `signals` 关键字，在 `signals` 前面不能使用 `public` 、 `private` 和 `protected` 等限定符，因为只有定义该信号的类及其子类才可以发射该信号。而且信号只用声明，不需要也不能对它进行定义实现。还要注意，信号没有返回值，只能是 `void` 类型的。因为只有 `QObject` 类及其子类派生的类才能使用信号和槽机制，这里的 `MyDialog` 类继承自 `QDialog` 类， `QDialog` 类又继承自 `QWidget` 类， `QWidget` 类是 `QObject` 类的子类，所以这里可以使用信号和槽。不过，使用信号和槽，还必须在类声明的最开始处添加 `Q_OBJECT` 宏，在这个程序中，类的声明是自动生成的，已经添加了这个宏。

在 `mydialog.ui` 对应的界面中添加一个 `Spin Box` 部件和一个 `Push Button` 部件，将 `pushButton` 的显示文本改为“确定”。然后转到 `pushButton` 的单击信号 `clicked()` 槽，更改如下：

```
void MyDialog::on_pushButton_clicked()   // 确定按钮
{
    int value = ui->spinBox->value();    // 获取输入的数值
    emit dlgReturn(value);               // 发射信号
    close();                             // 关闭对话框
}
```

当单击确定按钮时，便获取 `spinBox` 部件中的数值，然后使用自定义的信号将其作为参数发射出去。发射一个信号要使用 `emit` 关键字，例如程序中发射了 `dlgReturn()` 信号。

然后到 `widget.h` 文件中添加自定义槽的声明：

```
private slots:
    void showValue(int value);
```

声明一个槽需要使用 `slots` 关键字。一个槽可以是 `private` 、 `public` 或者 `protected` 类型的，槽也可以被声明为虚函数，这与普通的成员函数是一样的，也可以像调用一个普通函数一样来调用槽。槽的最大特点就是可以和信号关联。

下面打开 `widget.ui` 文件，向界面上拖入一个 `Label` 部件，然后更改其文本为“获取的值是：”。然后进入 `widget.cpp` 文件中添加头文件 `#include "mydialog.h"` ，再在构造函数中添加代码：

```
MyDialog *dlg = new MyDialog(this);
// 将对话框中的自定义信号与主界面中的自定义槽进行关联
connect(dlg,SIGNAL(dlgReturn(int)),this,SLOT(showValue(int)));
dlg->show();
```

这里创建了一个 `MyDialog` ，并且使用 `Widget` 作为父部件。然后将 `MyDialog` 类的 `dlgReturn()` 信号与 `Widget` 类的 `showValue()` 槽进行关联。信号和槽进行关联，使用的是 `QObject` 类的 `connect()` 函数，这个函数的原型如下：

```
bool QObject::connect ( const QObject *sender, const char * signal, const QObject * re
ceiver, const char * method,Qt::ConnectionType type = Qt::AutoConnection )
```

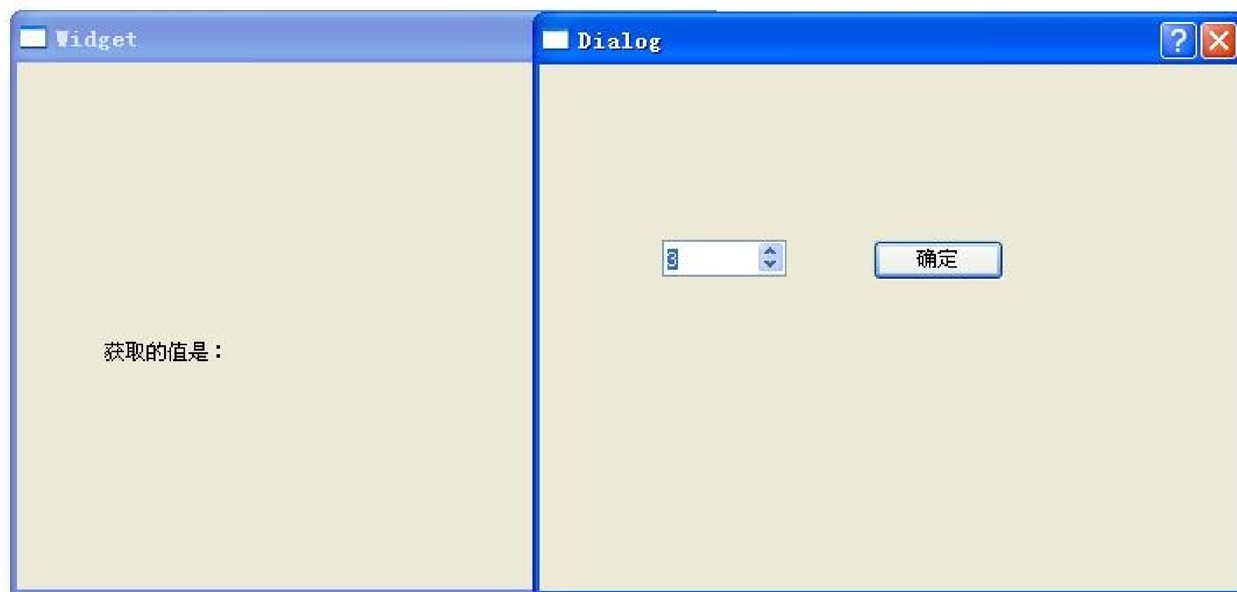
它的第一个参数为发送信号的对象，例如这里的 `dlg` ；第二个参数是要发送的信号，这里是 `SIGNAL(dlgReturn(int))` ；第三个参数是接收信号的对象，这里是 `this` ，表明是本部件，即 `Widget` ，当这个参数为 `this` 时，也可以将这个参数省略掉，因为 `connect()` 函数还有另外一个重载形式，该参数默认为 `this` ；第四个参数是要执行的槽，这里是 `SLOT(showValue(int))` 。对于信号和槽，必须使用 `SIGNAL()` 和 `SLOT()` 宏，它们可以将其参数转化为 `const char*` 类型。 `connect()` 函数的返回值为 `bool` 类型，当关联成功时返回 `true` 。还要注意，在调用这个函数时信号和槽的参数只能有类型，不能有变量，例如写

成 `SLOT(showValue(int value))` 是不对的。对于信号和槽的参数问题，基本原则是信号中的参数类型要和槽中的参数类型相对应，而且信号中的参数可以多于槽中的参数，但是不能反过来，如果信号中有多余的参数，那么它们将被忽略。下面介绍一下 `connect()` 函数的最后一个参数，它表明了关联的方式，其默认值是 `Qt::AutoConnection` ，这里还有其他几个选择，在编程中一般使用默认值，例如这里，在MyDialog类中使用emit发射了信号之后，就会执行槽，只有等槽执行完了以后，才会执行emit语句后面的代码。大家也可以将这个参数改为 `Qt::QueuedConnection` ，这样在执行完 `emit` 语句后便会立即执行其后面的代码，而不管槽是否已经执行。当不再使用这个关联时，还可以使用 `disconnect()` 函数来断开关联。

下面是自定义槽的实现，在这里只是简单的将参数传递来的数值显示在了标签上。因为这里使用了中文，所以大家记着在 `main.cpp` 文件中添加相关代码。

```
void Widget::showValue(int value)         // 自定义槽
{
    ui->label->setText(tr("获取的值是：%1").arg(value));
}
```

现在大家可以运行一下程序查看效果。如下图所示。



这个程序中自定义了信号和槽，可以看到它们的使用是很简单的，只需要对它们进行关联，然后在适当的时候发射信号就行。下面列举一下使用信号和槽应该注意的几点：

需要继承自 `QObject` 或其子类；

在类声明的最开始处添加 `Q_OBJECT` 宏；

槽中的参数的类型要和信号的参数的类型相对应，且不能比信号的参数多；

信号只用声明，没有定义，且返回值为 `void` 类型。

二、信号和槽的自动关联

信号和槽还有一种自动关联方式，例如前面程序中在设计模式直接生成的按钮的单击信号的槽，就是使用的这种方式：`on_pushButton_clicked()`，它由“on”、部件的`objectName`和信号三部分组成，中间用下划线隔开。这样组织的名称的槽就可以直接和信号关联，而不用再使用connect()函数。不过使用这种方式还要进行其他设置，而前面之所以可以直接使用，是因为程序中默认已经进行了设置。下面来看一个简单的例子。

新建Qt Gui应用，项目名称为`mySignalSlot2`，基类选择`QWidget`，然后类名保持`Widget`不变。完成后先在`widget.h`文件中进行函数声明：

```
private slots:
    void on_myButton_clicked();
```

这里自定义了一个槽，它使用自动关联。然后在`widget.cpp`文件中添加头文件`#include <QPushButton>`，再将构造函数的内容更改如下：

```
Widget::Widget(QWidget *parent) :
   QWidget(parent),
   ui(new Ui::Widget)
{
   QPushButton *button = new QPushButton(this); // 创建按钮
   button->setObjectName("myButton");          // 指定按钮的对象名
   ui->setupUi(this);                        //要在定义了部件以后再调用这个函数
}
```

因为在`setupUi()`函数中调用了`connectSlotsByName()`函数，所以要使用自动关联的部件的定义都要放在`setupUi()`函数之前，而且还必须使用`setObjectName()`函数指定它们的`objectName`，只有这样才能正常使用自动关联。下面是槽的定义：

```
void Widget::on_myButton_clicked()          // 使用自动关联
{
   close();
}
```

这里进行了关闭部件的操作。对于槽的函数名，中间要使用前面指定的`objectName`，这里是`myButton`。现在运行一下程序，单击按钮，发现可以正常关闭窗口。

可以看到，如果要使用信号和槽的自动关联，就必须在`connectSlotsByName()`函数之前进行部件的定义，而且还要指定部件的`objectName`。鉴于这些约束，虽然自动关联形式上很简单，但是实际编写代码时却很少使用。而且，在定义一个部件时，很希望明确的使用`connect()`函数来对其进行信号和槽的关联，这样当别人看到这个部件定义时，就可以知道和它相关的信号和槽的关联了。而使用自动关联，却没有这么明了。

三、信号和槽的高级应用

有时我们希望获得信号发送者的信息，在Qt中提供了`QObject::sender()`函数来返回发送该信号的对象的指针。但是如果有多个信号关联到了同一个槽上，而在该槽中需要对每一个信号进行不同的处理，使用上面的方法就很麻烦了。对于这种情况，便可以使

用 `QSignalMapper` 类。 `QSignalMapper` 可以被叫做信号映射器，它可以实现对多个相同部件的相同信号进行映射，为其添加字符串或者数值参数，然后再发射出去。对于这个类的使用，大家可以参考《Qt及Qt Quick开发实战精解》的1.3.3小节，那里有这个类的实际应用。还有就是Qt的演示程序中的 `Tools` 分类下的 `Input Panel` 示例程序中也使用了这个类，大家也可以参考一下这个程序。在这里便不再详细讲述这个类的使用了。

在本节的最后，来看一下信号和槽机制的特色和优越性：

- 信号和槽机制是类型安全的，相关联的信号和槽的参数必须匹配；
- 信号和槽是松耦合的，信号发送者不知道也不需要知道接受者的信息；
- 信号和槽可以使用任意类型的任意数量的参数。

结语

虽然信号和槽机制提供了高度的灵活性，但就其性能而言，还是慢于回调机制的。当然，这点性能差异通常在一个应用程序中是很难体现出来的。

涉及到的代码:

- mySignalSlot.rar
- mySignalSlot2.rar

第45篇 进阶（五）Qt样式表

导语

一个完善的应用程序不仅应该有实用的功能，还要有一个漂亮的外观，这样才能使应用程序更加友善，更加吸引用户。作为一个跨平台的UI开发框架，Qt提供了强大而灵活的界面外观设计机制。 Qt样式表是一个可以自定义部件外观的十分强大的机制。Qt样式表的概念、术语和语法都受到了HTML的层叠样式表（Cascading StyleSheets，CSS）的启发，不过与CSS不同的是，Qt样式表应用于部件的世界。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

正文

一、简介

要学习Qt样式表，需要对其有一个全面的了解，应该知道它到底有什么用，可以给哪些部件设置样式。为了了解这些内容，我们先在Qt帮助中查看Qt StyleSheets关键字，如下图所示。

Qt Style Sheets

 Home　　Qt Style Sheets

Qt Style Sheets are a powerful mechanism that allows you to c
subclassing QStyle. The concepts, terminology, and syntax of Qt
adapted to the world of widgets.

Topics:

- Overview
- The Style Sheet Syntax
- Qt Designer Integration
- Customizing Qt Widgets Using Style Sheets
- Qt Style Sheets Reference
- Qt Style Sheets Examples

Overview

Styles sheets are textual specifications that can be set on the who
several style sheets are set at different levels, Qt derives the effectiv

这里将所有内容分为了几部分：The Style Sheet Syntax中介绍了Qt样式表的语法，就是一些使用规则；Qt Designer Integration中介绍了如何在设计器中使用Qt样式表；CustomizingQt Widgets Using Style Sheets中介绍了如何使用Qt样式表来定制部件样式；QtStyle Sheets Reference中罗列了Qt中所有可以使用样式表的部件；QtStyle Sheets Examples中列出了常用部件使用样式表的例子，这个是我们后面学习使用时的重要参考。

二、在设计模式使用Qt样式表

1．新建Qt Gui应用，项目名称为 `myStyle` ，其他保持默认即可。完成后打开 `mainwindow.ui` 进入设计模式，然后拖入一个 `Push Button` 按钮。

2．在按钮部件上右击，选择“改变样式表”菜单项，在弹出的编辑样式表对话框中点击“添加颜色”下拉框，然后选择 `background-color` ，我们为其添加背景颜色。如下图所示。



这时会弹出选择颜色按钮，大家可以随便选择一个颜色，这里选择了红色，然后点击确定按钮关闭对话框。添加好的代码如下图所示。这种方法可以快速设置样式表，当然我们也可以自己手动来添加代码。



3．完成后大家可能发现按钮的颜色并没有改变，不要着急，这时运行程序，发现已经有效果了。如下图所示。



4・其实在设计模式还可以很容易地使用背景图片，这个需要使用Qt资源，大家可以试试，这里就不再介绍。

三、在代码中设置Qt样式表

既然在设计器中可以使用样式表，那么使用代码就一定可以实现。在代码中可以使用 `setStyleSheet()` 函数来设置样式表，不过用两种设置方法。

1・设置所有的相同部件都使用相同的样式。我们在 `mainwindow.cpp` 的构造函数中添加如下代码： `setStyleSheet("QPushButton { color: white }");`

这时运行程序，效果如下图所示。



可以看到按钮的文本颜色变成了白色，不过这种方式是给所有`QPushButton`类对象设置的样式。也就是说，我们再往界面上拖放其他的`Push Button`，它的文本颜色也会变成白色。

2．那么怎样才能只给特定的一个按钮设置样式表呢，这就需要使用第二种方式了。我们接着在`mainwindow.cpp`构造函数中添加代码：

```
ui->pushButton->setStyleSheet("color: blue");
```

这样就是只给先前添加的`pushButton`按钮设置了样式，将文本颜色设置为蓝色。为了有一个对比，大家可以再往界面上拖入一个`Push Button`按钮，然后运行程序，如下图所示。



也许现在又会问了，怎么按钮的背景不是红色的了？这是因为一个部件只能单独设置一个样式表，我们在代码中为`pushButton`设置了样式表就会屏蔽设计器中设置的。这里只是说单独为一个部件同时设置了多个样式表会出现这种情况，如果对其父类进行设置，则只会对其有影响，但是不会屏蔽掉自己的样式表，比如前面按钮的红底白字就是这种情况。

下面我们把代码更改如下：

```
ui->pushButton->setStyleSheet("background-color:red; color: blue");
```

再次运行程序，可以发现已经是红底蓝字了。效果如下图所示。



现在大家应该可以了解到，我们前面在设计模式中就是只为指定的 `pushButton` 按钮设置了背景。

四、样式表语法

1、样式规则

样式表包含了一系列的样式规则，一个样式规则由一个选择符（selector）和一个声明（declaration）组成。选择符指定了受该规则影响的部件；声明指定了这个部件上要设置的属性。例如：

```
QPushButton{color:red}
```

在这个样式规则中，`QPushButton` 是选择符，`{color:red}` 是声明，而 `color` 是属性，`red` 是值。这个规则指定了 `QPushButton` 和它的子类应该使用红色作为它们的前景色。Qt样式表中一般不区分大小写，例如 `color`、`Color`、`COLOR` 和 `ColoR` 表示相同的属性。只有类名，对象名和Qt属性名是区分大小写的。一些选择符可以指定相同的声明，只需要使用逗号隔开，例如：

```
QPushButton,QLineEdit,QComboBox{color:red}
```

一个样式规则的声明部分是一些属性：值对组成的列表，它们包含在大括号中，使用分号隔开。例如：

```
QPushButton{color:red;background-color:white}
```

2、子控件（Sub-Controls）

对一些复杂的部件修改样式，可能需要访问它们的子控件，例如`QComboBox`的下拉按钮，还有`QSpinBox`的向上和向下的箭头等。选择符可以包含子控件来对部件的特定子控件应用规则，例如：

```
QComboBox::drop-down{image:url(dropdown.png)}
```

这样的规则可以改变所有的`QComboBox`部件的下拉按钮的样式。在Qt Style Sheets Reference关键字对应的帮助文档的List of Stylable Widgets一项中列出了所有可以使用样式表来自定义样式的Qt部件，在List of Sub-Controls一项中列出了所有可用的子控件。

3．伪状态（Pseudo-States）

选择符可以包含伪状态来限制规则在部件的指定的状态上应用。伪状态出现在选择符之后，用冒号隔离，例如：

```
QPushButton:hover{color:white}
```

这个规则表明当鼠标悬停在一个`QPushButton`部件上时才被应用。伪状态可以使用感叹号来表示否定，例如要当鼠标没有悬停在一个`QRadioButton`上时才应用规则，那么这个规则可以写为：

```
QRadioButton:!hover{color:red}
```

伪状态还可以多个连用，达到逻辑与效果，例如当鼠标悬停在一个被选中的`QCheckBox`部件上时才应用规则，那么这个规则可以写为：

```
QCheckBox:hover:checked{color:white}
```

如果有需要，也可以使用逗号来表示逻辑或操作，例如：

```
QCheckBox:hover,QCheckBox:checked{color:white}
```

在Qt Style Sheets Reference关键字对应的帮助文档的List of Pseudo-States一项中列出了Qt支持的所有的伪状态。

4．例子

大家可以在Qt Style Sheets Examples页面找到很多相关的例子来学习，例如，下面是`QSpinBox`部件的一段样式表：

```
QSpinBox {
     padding-right: 15px; /* make room for the arrows */
     border-image: url(:/images/frame.png) 4;
     border-width: 3;
}
QSpinBox::up-button {
     subcontrol-origin: border;
     subcontrol-position: top right; /* position at the top right corner */
     width: 16px; /* 16 + 2*1px border-width = 15px padding + 3px parent border */
     border-image: url(:/images/spinup.png) 1;
     border-width: 1px;
}
QSpinBox::up-button:hover {
     border-image: url(:/images/spinup_hover.png) 1;
}
QSpinBox::up-button:pressed {
     border-image: url(:/images/spinup_pressed.png) 1;
}
QSpinBox::up-arrow {
     image: url(:/images/up_arrow.png);
     width: 7px;
     height: 7px;
}
QSpinBox::up-arrow:disabled, QSpinBox::up-arrow:off { /* off state when value is max */

    image: url(:/images/up_arrow_disabled.png);
}
QSpinBox::down-button {
     subcontrol-origin: border;
     subcontrol-position: bottom right; /* position at bottom right corner */
     width: 16px;
     border-image: url(:/images/spindown.png) 1;
     border-width: 1px;
     border-top-width: 0;
}
QSpinBox::down-button:hover {
     border-image: url(:/images/spindown_hover.png) 1;
}
QSpinBox::down-button:pressed {
     border-image: url(:/images/spindown_pressed.png) 1;
}
QSpinBox::down-arrow {
     image: url(:/images/down_arrow.png);
     width: 7px;
     height: 7px;
}
QSpinBox::down-arrow:disabled,
QSpinBox::down-arrow:off { /* off state when value in min */
    image: url(:/images/down_arrow_disabled.png);
}
```

结语

要想为软件设计一个漂亮的界面，需要灵活使用Qt样式表，不过这需要一定的CSS功底，还需要有美工经验。这一节只是简单介绍了下Qt中样式表的应用，只为抛砖引玉。大家也可以参考《QtCreator快速入门》第8章的相关内容，里面还涉及到了换肤、透明窗体、不规矩窗体等内容。

涉及到的代码

第46篇 进阶（六） 国际化

导语

在第2篇中讲述如何显示中文时，曾提到使用`QTextCodec`和`tr()`的方式直接显示中文，其实这只是一种临时的方法，方便我们快速完成程序，显示效果。当眞正要发布一个程序时，最好的方式是在程序中使用英文字符串，而后使用国际化工具进行翻译。

国际化的英文表述为Internationalization，通常简写为I18N（首尾字母加中间的字符数），一个应用程序的国际化就是使该应用程序可以让其他国家的用户使用的过程。Qt支持现在使用的大多数语言，特别是：

- 所有东亚语言（汉语、日语和朝鲜语）
- 所有西方语言（使用拉丁字母）
- 阿拉伯语
- 西里尔语言（俄语和乌克兰语等）
- 希腊语
- 希伯来语
- 泰语和老挝语
- 所有在Unicode5.1中不需要特殊处理的脚本

在Qt中，所有的输入部件和文本绘制方式对Qt所支持的所有语言都提供了内置的支持。Qt内置的字体引擎可以在同一时间正确而且精细的绘制不同的文本，这些文本可以包含来自众多不同书写系统的字符。

在Qt中可以使用Qt Linguist工具来很容易的完成应用程序的翻译工作，在Qt中编写代码时要对需要显示的字符串调用`tr()`函数，完成代码编写后，对这个应用程序的翻译主要包含三步：

- 运行lupdate工具从C++源代码中提取要翻译的文本，这时会生成一个`.ts`文件，这个文件是XML格式的；
- 在Qt Linguist中打开`.ts`文件，并完成翻译工作；
- 运行lrelease工具从`.ts`文件中获得`.qm`文件，它是一个二进制文件。这里的`.ts`文件是供翻译人员使用的，而在程序运行时只需要使用`.qm`文件，这两个文件都是与平台无关的。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目 录

- 二、更改项目文件
- 三、使用lupdate生成 `.ts` 文件
- 四、使用QtLinguist完成翻译
- 五、使用lrelease生成 `.qm` 文件
- 六、使用 `.qm` 文件

正文

一、编写源码

1．新建Qt Gui应用，项目名称为 `myI18N` ，类名为 `MainWindow` ，基类保持 `QMainWindow` 不变。

2．建立完项目后，点击 `mainwindow.ui` 文件进入设计模式，先添加一个 `&File` 菜单，再为其添加一个 `&New` 子菜单并设置快捷键为 `Ctrl+N` （不会操作，查看这里），然后往界面上拖入一个 `Push Button` 。

3．下面我们再使用代码添加几个标签，打开 `mainwindow.cpp` 文件，添加头文件 `#include <QLabel>` ，然后在构造函数中添加代码：

```
QLabel *label = new QLabel(this);
label->setText(tr("hello Qt!"));
label->move(100,50);
QLabel *label2 = new QLabel(this);
label2->setText(tr("password","mainwindow"));
label2->move(100,80);
QLabel *label3 = new QLabel(this);
int id = 123;
QString name = "yafei";
label3->setText(tr("ID is %1,Name is %2").arg(id).arg(name));
label3->resize(150,12);
label3->move(100,120);
```

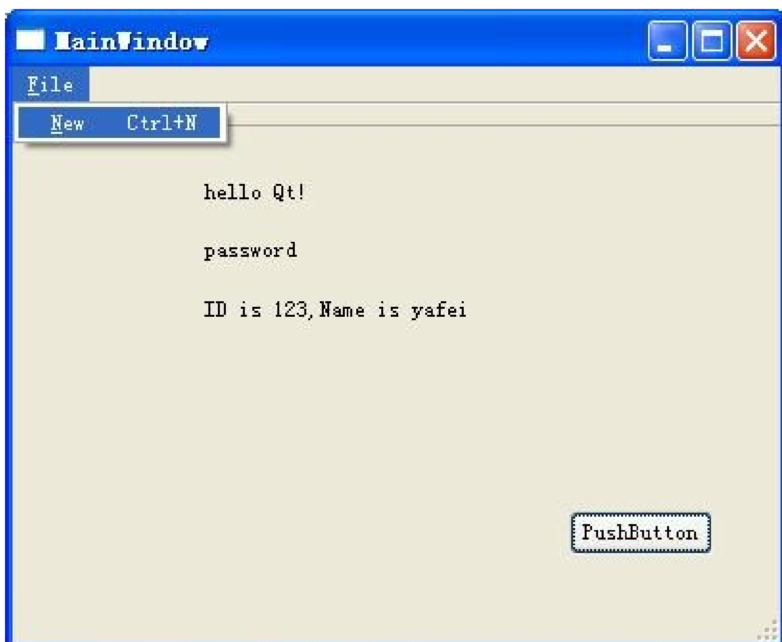
这里向界面上添加了三个标签，因为这三个标签中的内容都是用户可见的，所以需要调用 `tr()` 函数。在 `label2` 中调用 `tr()` 函数时，还使用了第二个参数，其实 `tr()` 函数一共有三个参数，它的原型如下：

```
QString QObject::tr( const char * sourceText, const char * disambiguation = 0, int n =
-1 )[static]
```

第一个参数 `sourceText` 就是要显示的字符串， `tr()` 函数会返回 `sourceText` 的译文；第二个参数 `disambiguation` 是消除歧义字符串，比如这里的 `password` ，如果一个程序中需要输入多个不同的密码，那么在没有上下文的情况下，就很难确定这个 `password` 到底指哪个密码。这个参数一般使用类名或者部件名，比如这里使用了 `mainwindow` ，就说明这个 `password` 是

在 `mainwindow` 上的；第三个参数n表明是否使用了复数，因为英文单词中复数一般要在单词末尾加“s”，比如“1 message”，复数时为“2 messages”。遇到这种情况，就可以使用这个参数，它可以根据数值来判断是否需要添加“s”。

4．运行程序效果如下图所示。



二、更改项目文件

我们要在项目文件中指定生成的 `.ts` 文件，每一种翻译语言对应一个 `.ts` 文件。打开 `myI18N.pro` 文件，在最后面添加如下一行代码：

```
TRANSLATIONS = myI18N_zh_CN.ts
```

这表明后面生成的 `.ts` 文件的文件名为 `myI18N_zh_CN.ts` ，对于 `.t` s的名称可以随意编写，不过一般是以区域代码来结尾，这样可以更好的区分，例如这里使用了“zh_CN”来表示简体中文。最后需要先按下 `Ctrl+S` 保存该文件。

三、使用lupdate生成 `.ts` 文件

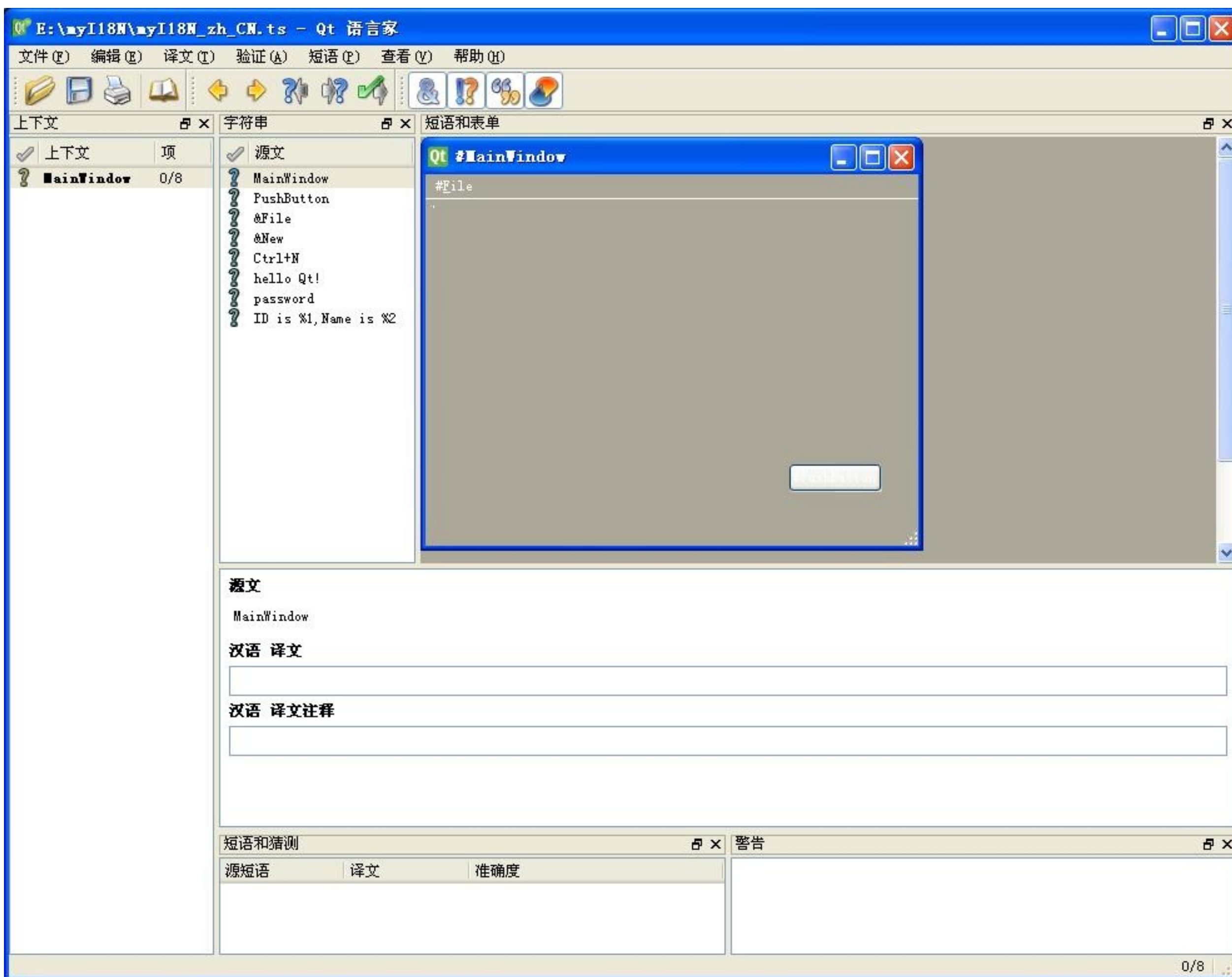
可以通过工具->外部->Qt语言家->更新翻译lupdate菜单项来完成该操作。这时会在概要信息处显示如下信息：

```
启动外部工具'C:/Qt/4.8.5/bin/lupdate.exe'E:/myI18N/myI18N.pro
C:/Qt/4.8.5/mkspecs/features/device_config.prf(13):Querying unknown property CROSS_COM
PILE
Updating 'myI18N_zh_CN.ts'...
Found 8 source text(s) (8 new and 0 already existing)
'C:/Qt/4.8.5/bin/lupdate.exe'完成
```

完成后可以在源码目录看到生成的 `myI18N_zh_CN.ts` 文件。

四、使用Qt Linguist完成翻译

这一步一般是翻译人员来做的，就是在Qt Linguist中打开 `.ts` 文件，然后对字符串逐个进行翻译。我们在系统的开始菜单中启动Linguist（也可以直接在命令行输入“linguist”启动它；或者在Qt安装目录的 `tools` 目录下找到并启动它），然后点击界面左上角的“打开”图标，在弹出的文件对话框中进入项目目录，打开 `myI18N_zh_CN` 文件，这时整个界面如下图所示。



下面来翻译程序。在翻译区域可以看到现在已经是要翻译成汉语，这是因为我们的 `.ts` 文件名中包含了中文的区域代码。如果这里没有正确显示要翻译成的语言，那么可以使用“编辑”→“翻译文件设置”菜单来更改。下面首先对 `MainWindow` 进行翻译，这里翻译为“应用程序主窗口”，然后按下 `Ctrl+Return` （即回车键）完成翻译并开始翻译第二个字符串。按照这种方法完成所有字符串的翻译工作，如下表所示。

```
原文本                翻译文本
MainWindow            应用程序主窗口
PushButton            按钮
&File                 文件(&F)
&New                  新建(&N)
Ctrl+N                Ctrl+N
hello Qt！            你好 Qt！
password              密码
ID is %1,Name is  %2  账号是%1，名字是%2
```

翻译完成后按下 `Ctrl+S` 保存更改，然后退出Qt Linguist。

五、使用lrelease生成 `.qm` 文件

可以通过工具->外部->Qt语言家->发布翻译lrelease菜单项来完成该操作。这时会在概要信息处显示如下信息：

```
启动外部工具'C:/Qt/4.8.5/bin/lrelease.exe'E:/myI18N/myI18N.pro
Updating 'E:/myI18N/myI18N_zh_CN.qm'...
Generated 8 translation(s) (8 finished and 0unfinished)
C:/Qt/4.8.5/mkspecs/features/device_config.prf(13):Querying unknown property CROSS_COM
PILE
'C:/Qt/4.8.5/bin/lrelease.exe'完成
```

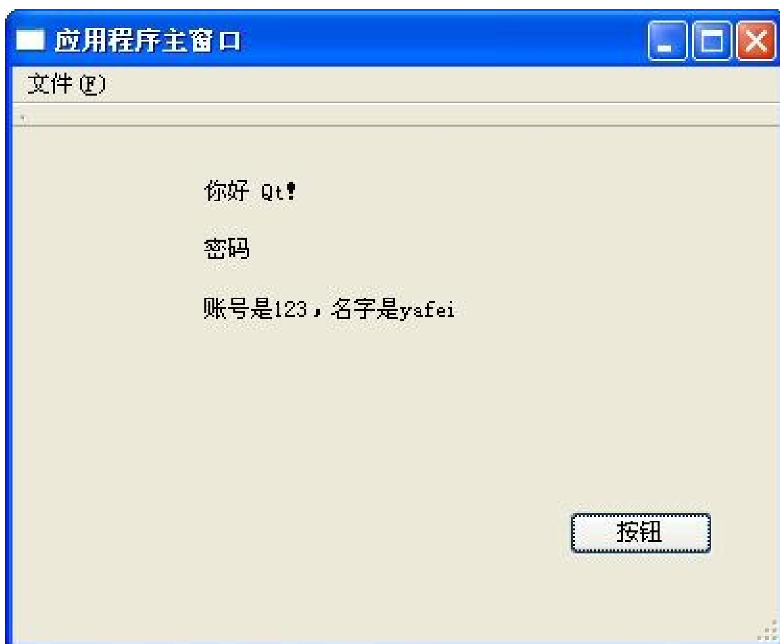
这时在源码目录会看到 `myI18N_zh_CN.qm` 文件。

六、使用 `.qm` 文件

下面在项目中添加代码使用 `.qm` 文件来更改界面的语言。进入 `main.cpp` 文件，添加头文件 `#include <QTranslator>` ，然后在 `QApplication a(argc, argv);` 代码下添加如下代码：

```
QTranslator translator;
translator.load("../myI18N/myI18N_zh_CN.qm");
a.installTranslator(&translator);
```

这里先加载了 `.qm` 文件（使用了相对路径），然后为 `QApplication` 对象安装了翻译。运行程序，效果下图所示。



结语

这一节简单介绍了一个使用Qt语言家实现国际化的例子，可以看到翻译一个程序其实是很简单的。Qt中还可以设置自动判断语言环境、动态进行语言更改等功能，详细内容可以参考帮助文档Internationalizationwith Qt或者参考《Qt Creator快速入门》第9章的相关内容。

涉及到的源码

第47篇 进阶（七） 定制Qt帮助系统

导语

一个完善的应用程序应该提供尽可能丰富的帮助信息。在Qt中可以使用工具提示、状态提示以及“What's This”等简单的帮助提示，也可以使用QtAssistant来提供强大的在线帮助。如果要进行详细的功能和使用的介绍，单单使用这些提示信息是不行的，这就需要提供HTML格式的帮助文本。在程序中可以通过调用Web浏览器或者使用`QTextBrowser`来管理和应用这些HTML文件。不过，Qt提供了更加强大的工具，那就是Qt Assistant，它支持索引和全文检索，而且可以为多个应用程序同时提供帮助，我们可以通过定制Qt Assistant来实现强大的在线帮助系统。

为了将Qt Assistant定制为我们自己的应用程序的帮助浏览器，需要先进行一些准备工作，主要是生成一些文件，最后再在程序中启动Qt Assistant。主要的步骤如下：

- 创建HTML格式的帮助文档；
- 创建Qt帮助项目（Qt help project）`.qhp`文件，该文件是XML格式的，用来组织文档，并且使它们可以在Qt Assistant中使用；
- 生成Qt压缩帮助（Qt compressed help）`.qch`文件，该文件由`.qhp`文件生成，是二进制文件；
- 创建Qt帮助集合项目（Qt help collection project）`.qhcp`文件，该文件是XML格式的，用来生成下面的`.qhc`文件；
- 生成Qt帮助集合（Qt help collection）`.qhc`文件，该文件是二进制文件，可以使Qt Assistant只显示一个应用程序的帮助文档，也可以定制Qt Assistant的外观和一些功能；
- 在程序中启动Qt Assistant。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

正文

一、创建HTML格式的帮助文档

1．新建Qt Gui应用，项目名称为 `myWhatsThis` ，类名为 `MainWindow` ，基类保持 `QMainWindow` 不变。

2．然后可以通过各种编辑器例如Microsoft Word来编辑要使用的文档，最后保存为HTML格式的文件，例如这里我们创建了5个HTML文件。然后在项目目录中新建文件夹，命名为 `documentation` ，再将这些HTML文件放入其中。再在 `documentation` 文件夹中再新建一个 `images` 文件夹，往里面复制一个图标图片，以后将作为Qt Assistant的图标，例如这里使用了 `yafeilinux.png` 图片。

二、创建 `.qhp` 文件

首先在 `documentation` 文件夹中创建一个文本文件，然后进行编辑，最后另存为 `myHelp.qhp` ，注意后缀为 `.qhp` 。文件的内容如下：

```
<?xml version="1.0"encoding="GB2312"?>
<QtHelpProjectversion="1.0">
<namespace>yafeilinux.myHelp</namespace>
<virtualFolder>doc</virtualFolder>
<filterSection>
   <toc>
     <section title="我的帮助"ref="./index.html">
       <section title="关于我们"ref="./aboutUs.html">
         <section title="关于yafeilinux"ref="./about_yafeilinux.html"></section>
         <section title="关于Qt Creator系列教程" ref="./about_QtCreator.html"></section>
       </section>
       <section title="加入我们"ref="./joinUs.html"></section>
     </section>
   </toc>
   <keywords>
     <keyword name="关于"ref="./aboutUs.html"/>
     <keyword name="yafeilinux" ref="./about_yafeilinux.html"/>
     <keyword name="Qt Creator"ref="./about_QtCreator.html"/>
   </keywords>
   <files>
     <file>about_QtCreator.html</file>
     <file>aboutUs.html</file>
     <file>about_yafeilinux.html</file>
     <file>index.html</file>
     <file>joinUs.html</file>
     <file>images/*.png</file>
   </files>
</filterSection>
</QtHelpProject>
```

这个 `.qhp` 文件是XML格式的，对于XML格式不是这里介绍的重点，我们主要讲解其中的内容，如果大家想了解XML格式相关的知识，可以参考这里。在第一行是XML序言，这里指定了编码 `encoding` 为GB2312，这样就可以使用中文了，如果只想使用英文，那么编码一般是UTF-8；第二行指定了 `QtHelpProject` 版本为1.0；第三行指定了命名空间 `namespace` ，每一个 `.qhp` 文件的命名空间都必须是唯一的，命名空间会成为Qt Assistant中的页面的URL的第一部分，这个在后面的内容中会涉及到；第四行指定了一个虚拟文件夹 `virtualFolder` ，这个文件夹并不需要创建，它只是用来区分文件的；再下面的过滤器部分filterSection标签包含了目录表、索引和所有文档文件的列表。过滤器部分可以设置过滤器属性，这样以后可以在Qt

Assistant中通过过滤器来设置文档的显示有否，不过，因为我们这里只有一个文档，所以不需要Qt Assistant的过滤器功能，这里也就不需要设置过滤器属性；在目录表`toc`（table ofcontents）标签中创建了所有HTML文件的目录，指定了它们的标题和对应的路径，这里设定的目录表为：

- 我的帮助
    - 关于我们
        - 关于yafeilinux
        - 关于Qt Creator系列教程
    - 加入我们

然后是`keywords`标签，它指定了所有索引的关键字和对应的文件，这些关键字会显示在Qt Assistant的索引页面；在`files`标签中列出了所有的文件，也包含图片文件。

三、生成`.qch`文件

这里为了测试创建的文件是否可用，可以先生成`.qch`文件，然后在QtAssistant中注册它。这样运行QtAssistant就会看到我们添加的文档了。不过，这一步不是必须的。我们打开命令行控制台，然后使用cd命令跳转到项目目录的`documentation`目录中，分别输入下面的命令后按下回车：

```
qhelpgeneratormyHelp.qhp –o myHelp.qch
assistant –registermyHelp.qch
```

要保证命令可以正常运行，前提是已经将Qt安装目录的`bin`目录的路径添加到了系统的`PATH`环境变量中。命令运行结果如下图所示。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>E:

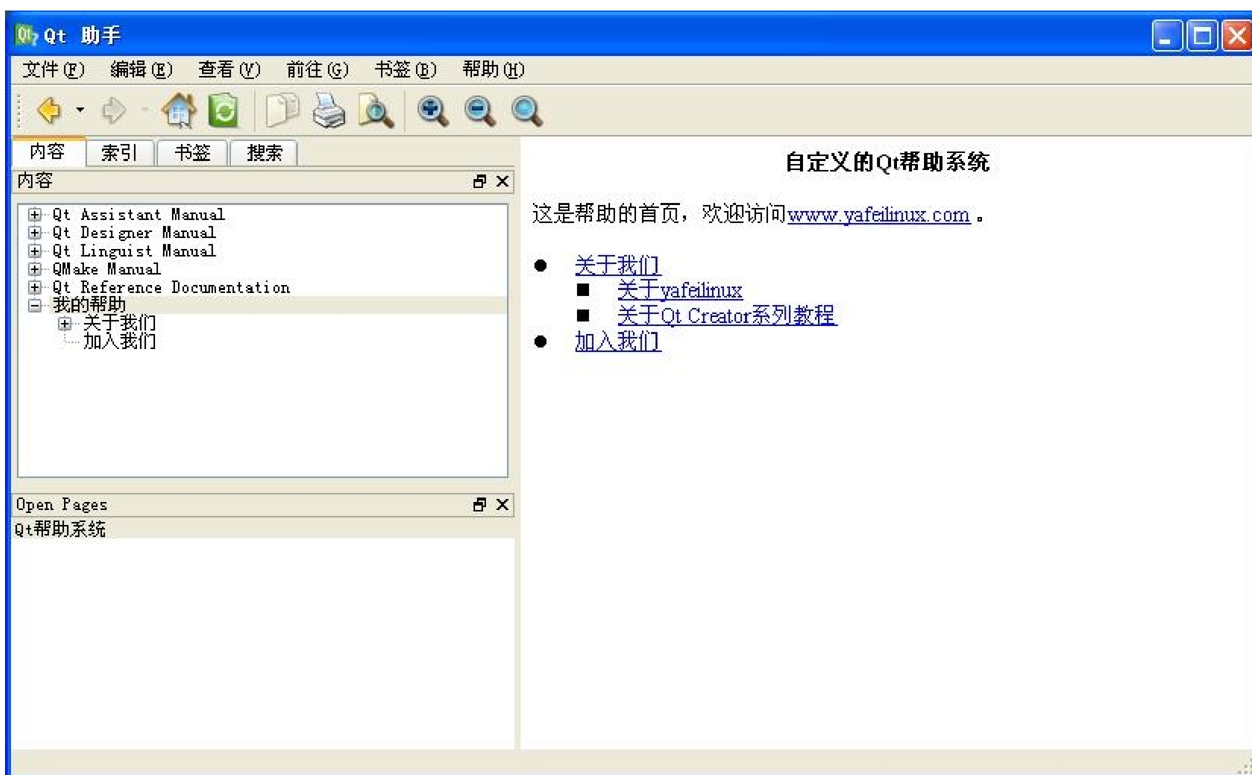
E:\>cd myWhatsThis\documentation

E:\myWhatsThis\documentation>qhelpgenerator myHelp.qhp -o myHelp.qch
Building up file structure...
Insert custom filters...
Insert help data for filter section (1 of 1)...
Insert files...
Insert contents...
Insert indices...
Documentation successfully generated.

E:\myWhatsThis\documentation>assistant -register myHelp.qch

E:\myWhatsThis\documentation>
```

当注册成功时，会显示“Documentationsuccessfully registered”提示对话框。这时在开始菜单中启动Qt Assistant（或者直接在命令行输入assistant来启动Qt Assistant），可以发现已经出现了我们的HTML文档，如下图所示。



四、创建 .qhcp 文件

要想使Qt Assistant只显示我们自己的帮助文档的最简单的方法就是生成帮助集合文件即 `.qhc` 文件，要生成 `.qhc` 文件，首先要创建 `.qhcp` 文件。在 `documentation` 文件夹中新建文本文档，对其进行编辑，最后另存为 `myHelp.qhcp` ，注意后缀为 `.qhcp` 。这里还要创建一个名为 `about.txt` 的文本文件，在其中输入一些该帮助的说明信息，作为QtAssistant的About菜单的显示内容。 `myHelp.qhcp` 文件的内容如下：

```
<?xml version="1.0"encoding="GB2312"?>
<QHelpCollectionProjectversion="1.0">
<assistant>
<title>我的帮助系统</title>
<applicationIcon>images/yafeilinux.png</applicationIcon>
<cacheDirectory>cache/myHelp</cacheDirectory>
<homePage>qthelp://yafeilinux.myHelp/doc/index.html</homePage>
<startPage>qthelp://yafeilinux.myHelp/doc/index.html</startPage>
<aboutMenuText>
   <text>关于该帮助</text>
</aboutMenuText>
<aboutDialog>
   <file>./about.txt</file>
   <icon>images/yafeilinux.png</icon>
</aboutDialog>
<enableDocumentationManager>false</enableDocumentationManager>
<enableAddressBar>false</enableAddressBar>
<enableFilterFunctionality>false</enableFilterFunctionality>
</assistant>
<docFiles>
<generate>
   <file>
     <input>myHelp.qhp</input>
     <output>myHelp.qch</output>
   </file>
</generate>
<register>
   <file>myHelp.qch</file>
</register>
</docFiles>
</QHelpCollectionProject>
```

在 `assistant` 标签中是对Qt Assistant的外观和功能的定制，其中设置了标题、图标、缓存目录、主页、起始页、About菜单文本、关于对话框的内容和图标等，还关闭了一些没有用的功能。对于缓存目录 `cacheDirectory` ，是进行全文检索等操作时缓存文件要存放的位置。对于主页 `homePage` 和起始页 `startPage` ，这里使用了第二步中提到的Qt Assistant的页面的URL，这个URL由 `qthelp://` 开始，然后是在 `.qhp` 文件中设置的命名空间，然后是虚拟文件夹，最后是具体的HTML文件名。因为Qt Assistant可以添加或者删除文档来为多个应用程序提供帮助，但是这里只是为一个应用程序提供帮助，并且不希望删除我们的文档，所以禁用了文档管理器 `documentation manager` ；因为这里的文档集很小，而且只有一个过滤器部分，所以也关闭了地址栏 `address bar` 和过滤器功能 `filter functionality` 。

在前面第三步中我们已经生成了 `.qch` 文件并且在Qt Assistant中进行了注册，但那只是为了测试文件是否可用，其实完全可以跳过第三步，因为在这里的 `docFiles` 标签中就完成了第三步的操作。不过与第三步不同的是，第三步是在默认的集合文件中注册的，而这里是在我们自己的集合文件中注册的。

五、生成 `.qhc` 文件

在命令行输入如下命令：

```
qcollectiongenerator myHelp.qhcp -omyHelp.qhc
```

输出结果如下图所示。

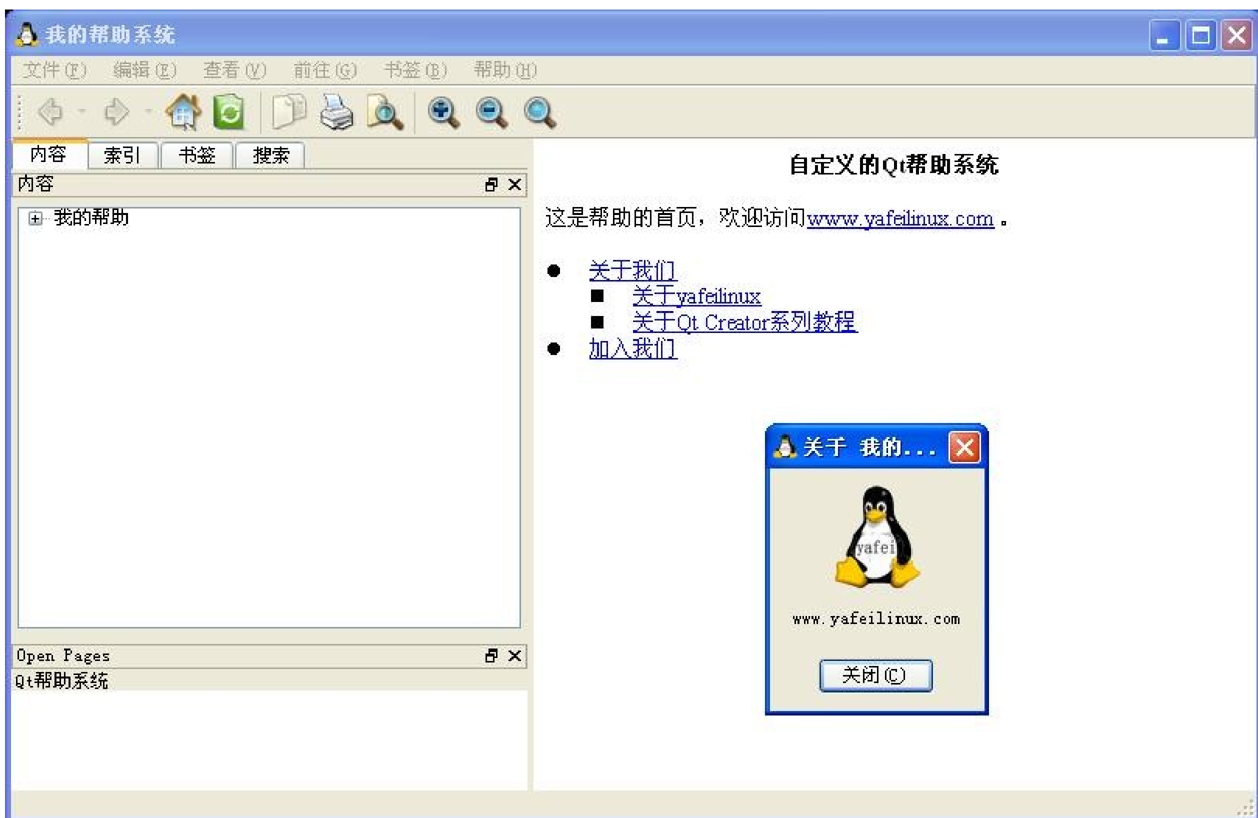
```
C:\WINDOWS\system32\cmd.exe
E:\myWhatsThis\documentation>qcollectiongenerator myHelp.qhcp -o myHelp.qhc
Reading collection config file...
Generating help for myHelp.qhp...
Building up file structure...
Insert custom filters...
Insert help data for filter section (1 of 1)...
Insert files...
Insert contents...
Insert indices...
Documentation successfully generated.
Creating collection file...

E:\myWhatsThis\documentation>_
```

为了测试我们定制的QtAssistant，可以在输入如下命令：

```
assistant -collectionFile myHelp.qhc
```

这里在运行Qt Assistant时指定了集合文件为我们自己的 `.qhc` 文件，所以运行后只会显示我们自己的HTML文档。可以看到，现在QtAssistant的图标也更改了，打开“Help”菜单中的“关于该帮助”菜单，这里是前面添加的 `about.txt` 文件的内容，效果如下图所示。



六、在程序中启动Qt Assistant

这里先要将Qt安装目录的 `bin` 目录中的 `assistant.exe` 程序复制到我们项目目录的 `documentation` 目录中。为了启动Qt Assistant，先要创建了一个 `Assistant` 类。首先向项目中添加新文件，模板选择C++ Class，类名为 `Assistant` ，基类不填写，类型信息选择无。然后将 `assistant.h` 文件更改如下：

```
#ifndef ASSISTANT_H
#define ASSISTANT_H
#include <QtCore/QString>
class QProcess;
class Assistant
{
public:
    Assistant();
    ~Assistant();
    void showDocumentation(const QString &file);
private:
    bool startAssistant();
    QProcess *proc;
};
#endif // ASSISTANT_H
```

在 `Assistant` 类中主要是使用 `QProcess` 类创建一个进程来启动Qt Assistant。下面是 `assistant.cpp` 文件的内容：

```
#include <QtCore/QByteArray>
#include <QtCore/QProcess>
#include <QtGui/QMessageBox>
#include "assistant.h"
Assistant::Assistant()
   : proc(0)
{
}
Assistant::~Assistant()
{
   if (proc && proc->state() == QProcess::Running) {
       // 试图终止进程
       proc->terminate();
       proc->waitForFinished(3000);
   }
   // 销毁proc
   delete proc;
}
// 显示文档
void Assistant::showDocumentation(const QString &page)
{
   if (!startAssistant())
       return;
   QByteArray ba("SetSource ");
   ba.append("qthelp://yafeilinux.myHelp/doc/");
   proc->write(ba + page.toLocal8Bit() + '\n');
}
// 启动Qt Assistant
bool Assistant::startAssistant()
{
   // 如果没有创建进程，则新创建一个
   if (!proc)
       proc = new QProcess();
   // 如果进程没有运行，则运行assistant，并添加参数
   if (proc->state() != QProcess::Running) {
     QString app = QLatin1String("../myWhatsThis/documentation/assistant.exe");
       QStringList args;
       args << QLatin1String("-collectionFile")
            << QLatin1String("../myWhatsThis/documentation/myHelp.qhc");
       proc->start(app, args);
       if (!proc->waitForStarted()) {
            QMessageBox::critical(0, QObject::tr("my help"),
                QObject::tr("Unable to launch Qt Assistant (%1)").arg(app));
            return false;
       }
   }
   return true;
}
```

在 `startAssistant()` 函数中使用 `QProcess` 创建了一个进程来启动Qt Assistant，这里使用了命令行参数来使用帮助集合文件，对于 `assistant.exe` 和 `myHelp.qhc` 都使用了相对地址；在 `showDocumentation()` 函数中可以指定具体的页面作为参数来使Qt Assistant显示指定的页面；在析构函数中，如果进程还在运行，则终止进程，最后销毁了进程指针。

下面来使用 `Assistant` 类来启动Qt Assistant。在 `mainwindow.h` 文件中先添加前置声明：

```
class Assistant;
```

再添加 `private` 对象指针声明：

```
Assistant *assistant;
```

然后添加一个私有槽：

```
private slots:
    void startAssistant();
```

现在到 `mainwindow.cpp` 文件中进行更改。添加头文件包含 `#include "assistant.h"` ，然后在构造函数中添加如下代码：

```
QAction *help = new QAction("help",this);
ui->mainToolBar->addAction(help);
connect(help, SIGNAL(triggered()), this, SLOT(startAssistant()));
// 创建Assistant对象
assistant = new Assistant;
```

这里创建了一个“help”动作，并将它添加到了工具栏中，可以使用该动作启动QtAssistant。下面是 `startAssistant()` 槽的定义：

```
void MainWindow::startAssistant()
{
   // 按下“help”按钮，运行Qt Assistant，显示index.html页面
   assistant->showDocumentation("index.html");
}
```

最后在析构函数中销毁 `assistant` 指针，即在 `MainWindow::~MainWindow()` 函数中添加如下代码：

```
// 销毁assistant
delete assistant;
```

现在运行程序，按下工具栏上的“help”动作，就可以启动Qt Assistant了。这里还要提示一下，如果要发布该程序，那么需要将 `documentation` 目录复制到发布目录中，这时运行程序。

结语

使用Qt定制帮助系统，可以制作功能强大的上下文相关的帮助文档，而对于一个优秀的软件而言，帮助菜单是必须有的。

涉及到的代码

第48篇 进阶（八）3D绘图简介

导语

OpenGL是一个跨平台的用来渲染3D图形的标准API。在Qt中提供了 `QtOpenGL` 模块，从而很轻松地实现了在Qt应用程序中使用OpenGL，这主要是在 `QGLWidget` 类中完成的。因为3D绘图涉及到了专业方面的内容，我们下面只是讲解最简单的使用，向大家演示在Qt中如何显示3D图形。如果大家想深入学习openGL绘图，可以查看网上比较经典的nehe的OpenGL教程，为了方便大家下载，我们在网站上提供了下载连接。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

正文

一、绘制简单的图形

`QGLWidget` 类是一个用来渲染OpenGL图形的部件，它提供了在Qt应用程序中显示OpenGL图形的功能。我们只需要继承该类，然后像使用其他 `QWidget` 部件一样来使用它。 `QGLWidget` 提供了三个虚函数，可以在子类中通过重新实现它们来执行典型的OpenGL任务：

- `initializeGL()` ：设置OpenGL渲染环境，定义显示列表等。该函数只在第一次调用 `resizeGL()` 或 `paintGL()` 前被调用一次；
- `resizeGL()` ：设置OpenGL的视口、投影等。每次部件改变大小时都会调用该函数；
- `paintGL()` ：渲染OpenGL场景。每当部件需要更新时都会调用该函数。

下面先来看一个简单的例子。

1．新建空的Qt项目，项目名称为 `myOpenGL` ，完成后向项目中添加新的C++ Class，类名为 `MyGLWidget` ，基类修改为 `QGLWidget` ，类型信息选择“继承自 `QWidget` ”。

2．完成后打开 `myOpenGL.pro` ，添加一行代码：

```
QT += opengl
```

然后保存该文件。

3．打开 `myglwidget.h` 文件，添加函数声明：

```
protected:
    void initializeGL();
    void resizeGL(int w, int h);
    void paintGL();
```

4．再到 `myglwidget.cpp` 文件中先添加头文件包含：

```
#include <GL/glu.h>
```

然后添加这三个函数的定义：

```
void MyGLWidget::initializeGL()
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glClearDepth(1.0);
   glEnable(GL_DEPTH_TEST);
}
```

`glClearColor()` 函数用来设置清除屏幕时使用的颜色，其四个参数分别用来设置红、绿、蓝颜色分量和Alpha值，它们的取值范围都是0.0到1.0，这里四个参数都为0，表示纯黑色。然后设置了阴影平滑（smooth shading），这样可以使色彩和光照更加精细。最后设置了深度缓存和启用深度测试，用来记录图形在屏幕内的深度值。

```
void MyGLWidget::resizeGL(int w, int h)
{
   glViewport(0, 0, (GLint)w, (GLint)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLfloat)w/(GLfloat)h, 0.1, 100.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

`glViewport()` 函数用来设置视口的大小。然后使用 `glMatrixMode()` 设置了投影矩阵，投影矩阵用来为场景增加，后面使用了 `glLoadIdentity()` 重置投影矩阵，这样可以将投影矩阵恢复到初始状态。 `gluPerspective()` 用来设置投影矩阵，这里设置视角为45度，纵横比为窗口的纵横比，最近的位置为0.1，最远的位置为100，这两个值是场景中所能绘制的深度的临界值。大家可以想象，离我们眼睛比较近的东西看起来比较大，而比较远的东西看起来就比较小。最后设置并重置了模型视图矩阵。

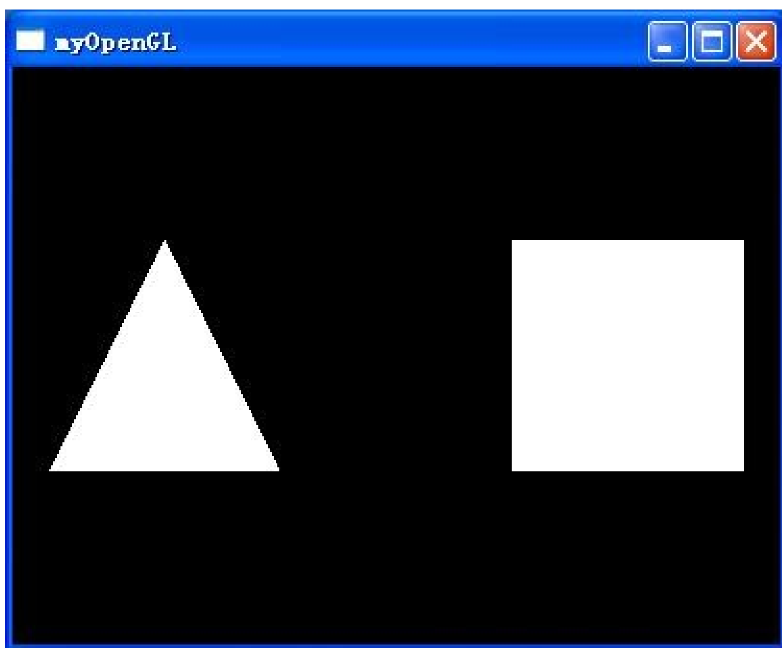
```
void MyGLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    // 绘制三角形
    glTranslatef(-2.0, 0.0, -6.0);
    glBegin(GL_TRIANGLES);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glEnd();
    // 绘制四边形
    glTranslatef(4.0, 0.0, 0.0);
    glBegin(GL_QUADS);
    glVertex3f(-1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
    glEnd();
}
```

在开始绘制以前，先要使用`glClear()`清除屏幕和深度缓存。然后重置了模型视图矩阵，这样便将当前点移动到了窗口的中心，现在窗口中心即为坐标原点，X轴从左到右，Y轴从下到上，Z轴从里到外。完成这两步以后就可以进行图形的绘制了，在图形绘制开始时，一般会使用`glTranslatef()`来移动坐标原点，它是相对于当前点来移动的，比如这里先将坐标原点左移2.0，向里移6.0，然后绘制了三角形（TRIANGLES）。绘制从`glBegin()`开始，到`glEnd()`结束，使用`glVertex3f()`来设置各个顶点的坐标，顶点的绘制顺序可以是顺时针，也可以是逆时针。要注意逆时针绘制出来的是正面，而顺时针绘制出来的是反面，这一点在后面的纹理贴图部分会显示出来。当绘制完三角形以后，又将原点相对于当前点向右移动了4.0，然后绘制了一个四边形（QUADS）。

5．最后再向项目中添加`main.cpp`文件，更改内容如下：

```
#include <QApplication>
#include "myglwidget.h"
int main(int argc, char *argv[])
{
    QApplication app(argc,argv);
    MyGLWidget w;
    w.resize(400, 300);
    w.show();
    return app.exec();
}
```

现在运行程序，效果如下图所示。



可以看到，在Qt中只需要实现这三个函数就可以进行OpenGL绘图了。

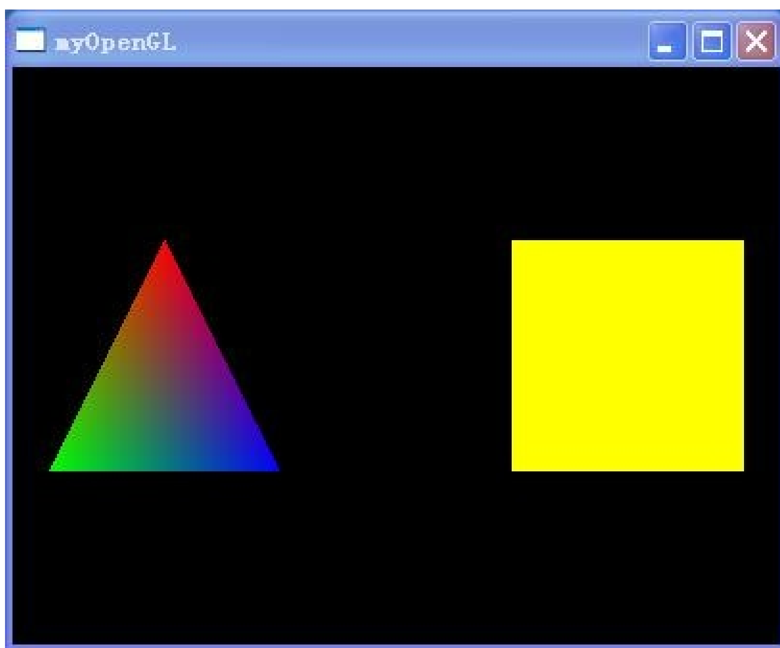
二、添加颜色

上面的例子中图形都是白色的，可以使用 `glColor3f()` 函数来设置绘制时使用的颜色，它的三个参数用来指定RGB（红绿蓝）颜色分量，取值范围为0.0到1.0。我们可以在绘制一个顶点时指定使用的颜色，如果后面不再设置其他颜色，那么所有的顶点都将使用同样的颜色。

下面我们将图形的绘制代码更改如下：

```
void MyGLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-2.0, 0.0, -6.0);
    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
    glColor3f(0.0, 0.0, 1.0);
    glVertex3f(1.0, -1.0, 0.0);
    glEnd();
    glTranslatef(4.0, 0.0, 0.0);
    glBegin(GL_QUADS);
    glColor3f(1.0, 1.0, 0.0);
    glVertex3f(-1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
    glEnd();
}
```

可以看到三角形的三个角分别是红色、绿色和蓝色，而正方形是纯黄色的。如下图所示。

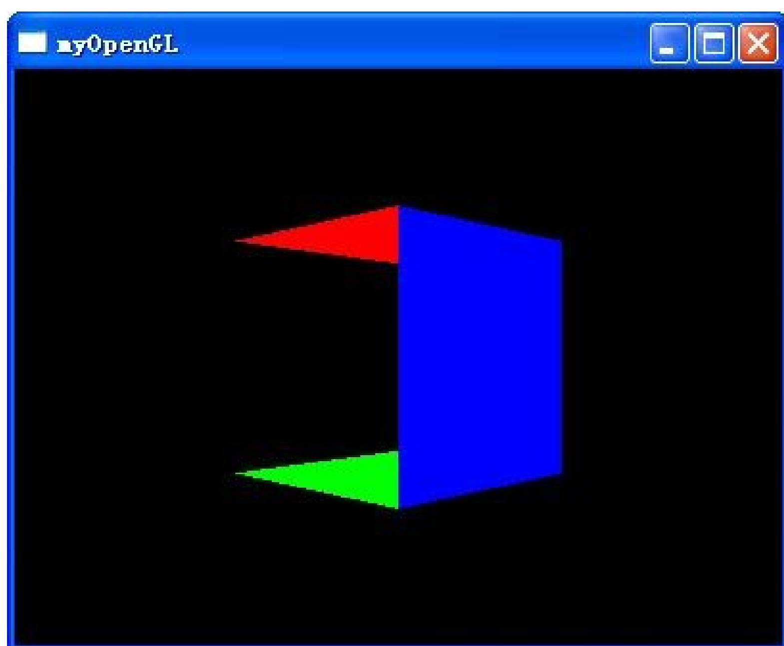


三、实现3D效果

前面的图形都还是平面图形，下面添加代码来实现三维立体效果。将程序中绘制图形的代码更改为：

```
void MyGLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -6.0);
    glRotatef(45, 0.0, 1.0, 0.0);
    glBegin(GL_QUADS);
    // 上面
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(1.0, 1.0, 1.0);
    glVertex3f(1.0, 1.0, -1.0);
    glVertex3f(-1.0, 1.0, -1.0);
    glVertex3f(-1.0, 1.0, 1.0);
    // 下面
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(1.0, -1.0, 1.0);
    glVertex3f(1.0, -1.0, -1.0);
    glVertex3f(-1.0, -1.0, -1.0);
    glVertex3f(-1.0, -1.0, 1.0);
    // 前面
    glColor3f(0.0, 0.0, 1.0);
    glVertex3f(1.0, 1.0, 1.0);
    glVertex3f(-1.0, 1.0, 1.0);
    glVertex3f(-1.0, -1.0, 1.0);
    glVertex3f(1.0, -1.0, 1.0);
    glEnd();
}
```

这里使用了 `glRotatef(angle, x, y, z)` 函数来对坐标轴进行旋转，它的第一个参数是旋转的角度，后面三个参数用来确定旋转轴向量。旋转轴经过原点，指向 `（x,y,z）` 点。图形的旋转满足右手定则，就是用右手握住旋转轴，大拇指指向旋转轴所指的方向，然后其他四个手指所指的方向就是要旋转的方向。再往下面，分别绘制了三个正方形作为正方体的三个面，大家还可以再补充上其他几个面。运行程序，效果如下图所示。



结语

这一节只是向大家简单演示了如何在Qt中进行openGL编程来实现3D绘图，目的只是让大家看到在Qt中进行3D绘图是非常简单的。如果想进一步应实现更炫酷的效果，就需要拥有openGL的专业知识了。

涉及到的代码

第49篇 进阶（九） 多媒体应用简介

导语

Qt对于音频视频的播放和控制等多媒体应用提供了强大的支持。要想使计算机发出响声，最简单的方法是调用 `QApplication::beep()` 静态函数；而对于简单的声音播放，可以使用 `QSound` 类；对于简单的动画播放，可以使用 `QMovie` 类；要想对音频视频实现更多的控制，可以使用Phonon多媒体框架；而对于音频视频底层的控制，可以使用 `QtMultimedia` 模块。

虽然在Qt 5中已经已经放弃了Phonon，但是使用Qt 4时，Phonon还是一个很好的选择，所以这里我们也对其进行了简单介绍。

环境：Windows Xp + Qt 4.8.5+QtCreator2.8.0

目录

正文

一、使用QSound播放声音

1．新建Qt Gui应用，名称为 `mySound` ，类名 `MainWindow` 和基类 `QMainWindow` 保持默认即可。

2．完成后在 `mainwindow.cpp` 文件中添加头文件 `#include <QSound>` ，然后在构造函数中添加如下一行代码：

```
QSound::play("../mySound/sound.wav");
```

这时运行程序就可以播放指定的音频文件了，注意这里将音频文件放在了项目目录中。因为现在 `QSound` 并不支持资源文件，所以音频文件必须要放在程序外面。除了简单的使用静态函数进行播放外，也可以先构建一个 `QSound` 对象，然后再调用 `play()` 槽进行播放，可以使用 `stop()` 槽来停止声音的播放，还可以使用 `setLoops()` 函数设置播放重复的次数，如果设置为-1表示无限循环。

3．先到 `mainwindow.h` 文件中添加前置声明 `classQSound;` 然后声明一个私有对象：

```
QSound *sound;
```

再到 `mainwindow.cpp` 文件中，将构造函数里添加的调用 `play()` 函数的代码更改为：

```
sound = new QSound("../mySound/sound.wav", this);
```

然后我们双击 `mainwindow.ui` 文件进入设计模式，向界面上添加两个 `Push Button` 和一个 `Spin Box` ，并将两个按钮的文本分别改为“播放”和“停止”。然后更改 `Spin Box` 的属性，

将最小值 `minimum` 设置为-1，将当前值value设置为1。最后分别转到两个按钮的 `clicked()` 槽和 `Spin Box` 的 `valueChanged(int )` 槽，更改它们的内容如下：

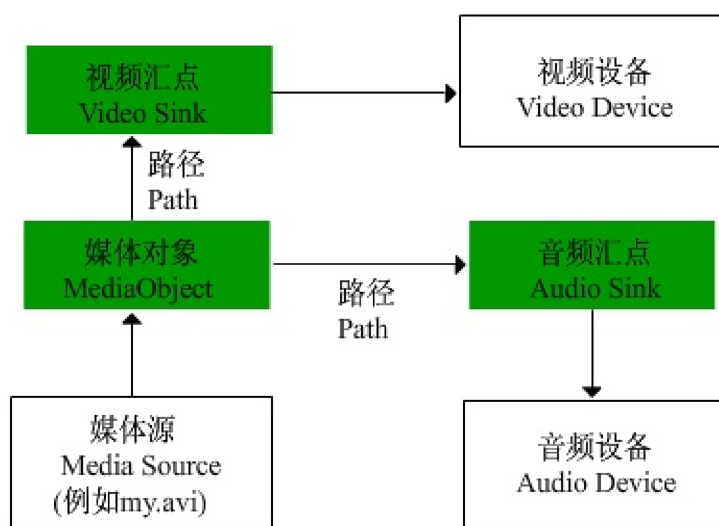
```
void MainWindow::on_pushButton_clicked()
{ // 播放按钮
    sound->play();
}
void MainWindow::on_pushButton_2_clicked()
{ // 停止按钮
    sound->stop();
}
void MainWindow::on_spinBox_valueChanged(int value)
{
    sound->setLoops(value);
}
```

4．现在运行程序，可以设置播放的次数，然后使用开始按钮进行播放，使用停止按钮来停止播放了。要说明一下，在Windows平台上，如果设置了循环次数，那么 `stop()` 函数无法立即停止播放，需要完成当前的循环才可以停止播放。

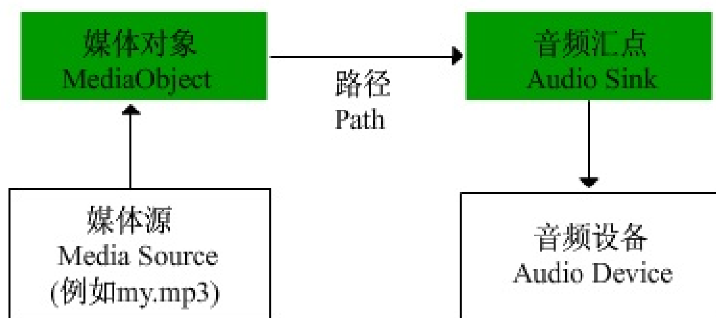
使用 `QSound` 可以实现一些简短的声音的播放，使用它来实现单击按钮或者其他事件的音效是很好的选择。根据平台的音频设备的不同，如果使用 `QSound` 同时播放多个音频文件，那么后面播放的声音或者会与前面播放的声音进行混合，或者会停止前面播放的声音，在Windows平台上，新播放的声音会停止前面播放的声音。如果要播放其他格式的音频文件，或者需要对播放进行更多的控制，那么就需要使用Phonon多媒体框架来实现了。

二、Phonon 简介

Phonon包含三个基本的概念：媒体对象（Media Objects）、汇点（Sinks）和路径（Paths）。媒体对象是 `MediaObject` 类的一个实例，用来管理媒体源，例如一个音乐文件，它提供开始、停止和暂停等简单的播放控制功能；汇点用来从Phonon中输出媒体，例如在部件上渲染视频或者将音频传送到声卡，汇点通常是一个渲染设备，如 `VideoWidget` ；路径用来连接Phonon对象，例如连接一个媒体对象和一个汇点，这样形成的关系图在Phonon中被称为媒体图（mediagraph）。下图是一个音频流的媒体图：



下图是一个播放带有声音的视频文件的媒体图：



多媒体功能并不是由Phonon本身实现的，而是由后端（也常被称为引擎）实现的，这包含了连接、管理和驱动底层的硬件或中间技术。对于编程人员来说，这意味着媒体的节点，例如媒体对象、处理器和汇点，它们都由后端提供。而且，后端还负责构建媒体图，例如连接各个节点。

Qt使用的后端，在Windows上是DirectShow，在Mac上是QuickTime，在Linux上是Gstreamer。依赖于底层的系统，在不同的平台上所提供的功能会有所不同。后端可以显示顶层系统的信息，可以获取它支持的媒体格式，例如AVI、mp3或者OGG等。

三、使用Phonon播放音频

当播放音频时，需要创建一个媒体对象，然后将它连接到一个音频输出节点，该节点由 `AudioOutput` 类提供，它用来将音频输出到声卡。

1．新建Qt Gui应用，名称为 `myPhonon1` ，类名 `MainWindow` 和基类 `QMainWindow` 保持默认即可。完成后，在项目文件 `myPhonon1.pro` 中添加如下一行代码：

```
QT        += phonon
```

然后保存该文件。

2．再到 `mainwindow.cpp` 文件中添加头文件包含：

```
#include <phonon>
#include <QDebug>
#include <QUrl>
```

并在构造函数中添加如下代码：

```
Phonon::MediaObject *mediaObject = new Phonon::MediaObject(this);
mediaObject->setCurrentSource(Phonon::MediaSource("../myPhonon1/mysong.mp3"));
Phonon::AudioOutput *audioOutput = new Phonon::AudioOutput(Phonon::MusicCategory, this
);
Phonon::Path path = Phonon::createPath(mediaObject, audioOutput);
mediaObject->play();
```

`AudioOutput` 类用来向音频输出设备发送数据，音频输出需要使用 `createPath()` 函数连接到媒体对象上。 `AudioOutput` 提供了函数来控制音量的大小，还可以设置静音，而且Phonon还使用AudioOutput提供了一个音量控制部件 `VolumeSlider` 类，它是 `QWidget` 的子类。另外，Phonon中也提供了播放进度滑块 `SeekSlider` ，该类也是QWidget的子类，它需要连接到媒体对象上。

3．往源码目录中添加一个 `mysong.mp3` 文件，运行程序就可以自动播放了。不过，现在还无法进行任何的控制。大家可以查看《Qt及Qt Quick开发实战精解》的音乐播放器实例，那是一个比较完整的播放器例子。

四、使用Phonon播放视频

1．新建Qt Gui应用，项目名称为 `myPhonon2` ，基类选择 `QWidget` ，类名为 `Widget` 。完成后先到项目文件 `myPhonon2.pro` 中添加代码：

```
QT       += phonon
```

然后保存该文件。

2．播放视频，可以使用 `VideoWidget` 类，该部件可以自动选择可用的设备来播放视频。 `VideoWidget` 并不会播放媒体流中的音频，如果要播放视频中的音频，那么就要创建一个 `AudioOutput` 节点。在 `mainwindow.cpp` 文件中添加头文件包含：

```
#include <phonon>
#include <QAction>
#include <QVBoxLayout>
#include <QToolBar>
```

再在构造函数中添加如下代码：

```
// 创建媒体图
Phonon::MediaObject *mediaObject = new Phonon::MediaObject(this);
Phonon::VideoWidget *videoWidget = new Phonon::VideoWidget(this);
Phonon::createPath(mediaObject, videoWidget);
Phonon::AudioOutput *audioOutput = new Phonon::AudioOutput(
                                    Phonon::VideoCategory, this);
Phonon::createPath(mediaObject, audioOutput);
mediaObject->setCurrentSource(Phonon::MediaSource(
"../myPhonon2/myVideo.WMV"));

// 创建播放进度滑块
Phonon::SeekSlider *seekSlider = new Phonon::SeekSlider(
mediaObject, this);

// 创建工具栏，包含了播放、暂停和停止动作，以及控制音量滑块
QToolBar *toolBar = new QToolBar(this);
QAction *playAction = new QAction(style()->standardIcon(QStyle::SP_MediaPlay), tr("播放"
), this);
connect(playAction, SIGNAL(triggered()), mediaObject, SLOT(play()));
QAction *pauseAction = new QAction(style()->standardIcon(QStyle::SP_MediaPause), tr("暂
停"), this);
connect(pauseAction, SIGNAL(triggered()), mediaObject, SLOT(pause()));
QAction *stopAction = new QAction(style()->standardIcon(QStyle::SP_MediaStop), tr("停止"
), this);
connect(stopAction, SIGNAL(triggered()), mediaObject, SLOT(stop()));
Phonon::VolumeSlider *volumeSlider = new Phonon::VolumeSlider(audioOutput, this);
volumeSlider->setSizePolicy(QSizePolicy::Maximum, QSizePolicy::Maximum);
toolBar->addAction(playAction);
toolBar->addAction(pauseAction);
toolBar->addAction(stopAction);
toolBar->addWidget(volumeSlider);

// 创建布局管理器，将各个部件都添加到布局管理器中
QVBoxLayout *mainLayout = new QVBoxLayout;
videoWidget->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
mainLayout->addWidget(videoWidget);
mainLayout->addWidget(seekSlider);
mainLayout->addWidget(toolBar);
setLayout(mainLayout);
```

这里创建了两个路径，分别用于视频流和音频流。创建 `VideoWidget` 时不需要指定类别，它会自动指定为 `VideoCategory` 类别，我们需要做的只是确保 `AudioOutput` 也在同一个类别中。Phonon中的 `SeekSlider` 类提供了一个滑块来定位媒体流，只要与媒体对象 `MediaObject` 连接后，该部件就可以自动和媒体流的播放进度同步起来，不需要我们手动进行信号和槽的关联。Phonon中的 `VolumeSlider` 部件提供了一个用来控制音频输出设备音量的滑块，该滑块还会默认显示一个可以设置静音的图标，这个图标可以使用 `setMuteVisible()` 来移除。

3、需要将一个 `myVideo.WMV` 的视频文件复制到源码目录，然后运行程序，效果如下图所示。



结语

`Movie` 类是一个使用 `QImageReader` 来播放动画的便捷类。该类用来显示没有声音的简单动画，主要支持GIF和MNG格式的文件；在Qt 4.6中新加入了 `QtMultimedia` 模块来提供一些底层的多媒体功能，比如音频的采集和回放、频谱分析、操作视频帧等。关于这两个类的使用，可以参考《Qt Creator 快速入门》的第13章，进一步学习Phonon的内容，可以参考第14章。

涉及到的源码:

- myPhonon1.rar
- myPhonon2.rar
- mySound.rar

第二部分 进入Qt 5的世界

（基于Qt 5，主要讲解Quick Qml编程）

QtQuick

过渡篇

从Qt 4到Qt 5（一）Qt 5.2安装、程序迁移和发布

导语

Qt 5的第二个重大版本Qt 5.2的beta版终于发布了，Qt 5.2是官方一再强调开发Android要使用的版本。经过了近一年的等待，这次终于可以完成夙愿，继续更新Qt系列教程了。在后面的教程中会尽量涉及大家经常问到、急需解决的问题，也会尽可能的把最新的技术和最炫的界面效果展示给大家。

这里也请大家把心态放平稳一些，是说大家学习的心态，也是说我写教程的心态。通过这几年的经历，我发现，凡事不能急功近利，只有平常心才能出眞知，只有用最朴实（有时候可能显得不专业）的语言来描述讲解一个问题，才会让更多人容易读懂，才会得到更多人的赞誉。这里不得不说，写教程只是我在业余时间做的事情，我的技术水平也没有一些网友想的那么牛叉，之所以还要一直写下去，是因为有那么多网友的支持和肯定。也是在今天，我得知《Qt Creator快速入门》已经售罄，这距该书出版还不到一年半的时间。这里再次谢谢那些支持我的朋友，我会通过更好的教程和开源作品来感谢大家一直以来的支持！

环境：Windows 7 + Qt 5.2.0+QtCreator 3.0

目录

内容概要

本节讲述的内容主要有三点：

第一，一般的Qt 4程序要在Qt 5上编译，需要注意：

1．将 `main.cpp` 文件中的 `#include <QtGui/QApplication>` 修改为 `#include <QApplication>`

2．在 `.pro` 项目文件中添加： `greaterThan(QT_MAJOR_VERSION, 4): QT += widgets`

第二，在Qt 5中设置应用程序图标，需要注意：

1．将 `.ico` 图标文件放到项目源码目录

2．在 `.pro` 文件中添加： `RC_ICONS = myico.ico` （ `myico.ico` 就是自己图标文件的名字）

第三，发布Qt 5程序时，除了必要的dll文件以外，还需要将`plugins`中的`platforms`目录复制过来，而里面只要保留`qminimal.dll`和`qwindows.dll`两个文件即可。

正文

一、软件安装

1．下载并安装Qt 5.2

首先到Qt官方下载页面：http://download.qt-project.org/development_releases/qt/5.2/5.2.0-beta1/

因为是在Windows下，所以下载含有Android库的Windows版本，具体文件是：

```
qt-windows-opensource-5.2.0-beta1-android-x86-win32-offline.exe
```

这个安装包中已经包含了所有需要的工具（例如最新版的Qt Creator 3.0，当然要开发Android还是需要自己添加文件的），我们只需要下载这一个文件即可。

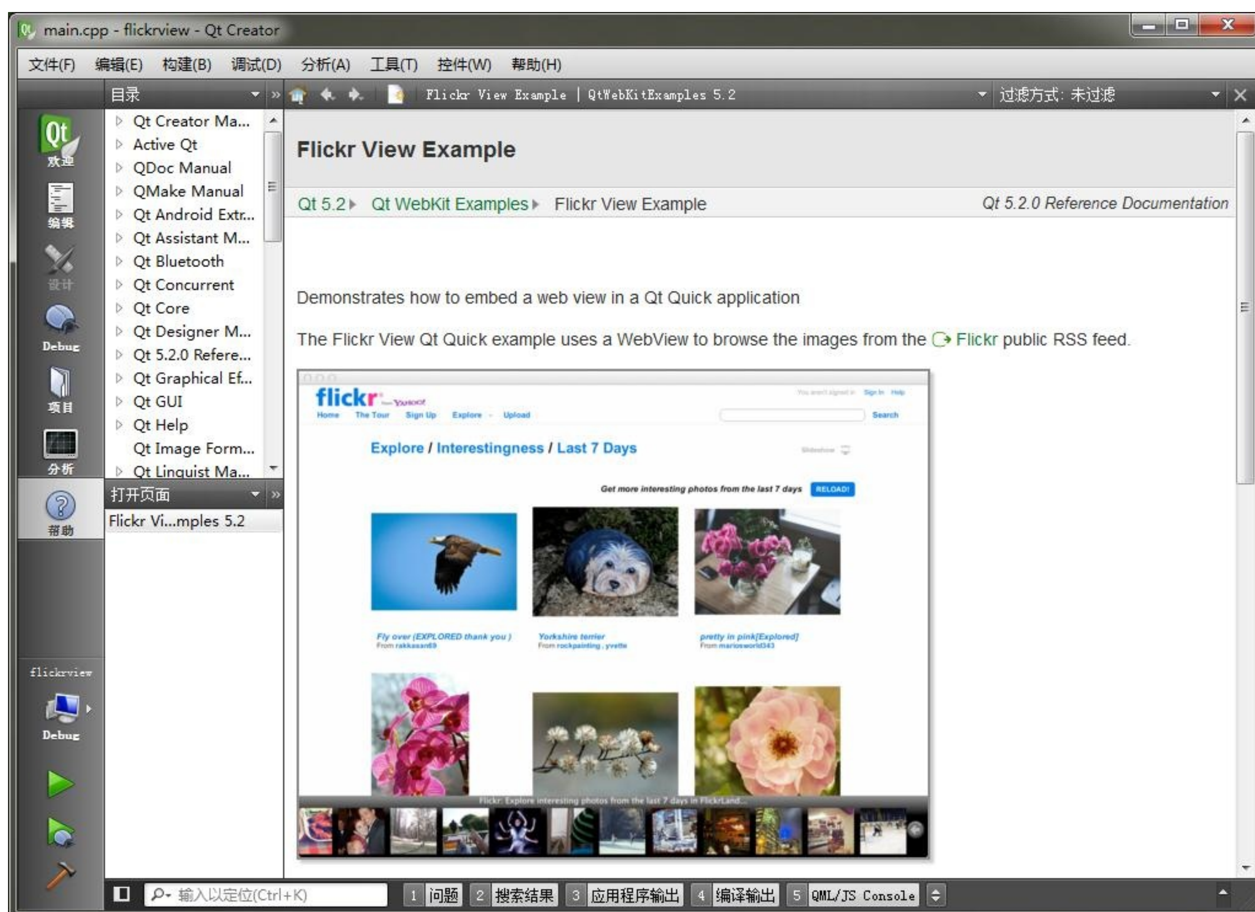
下载完成后，双击运行。这里一般不需要做任何设置，直接点击下一步直到软件安装完成。最后便自动打开了我们期盼已久的Qt Creator 3.0欢迎界面，如下图所示。



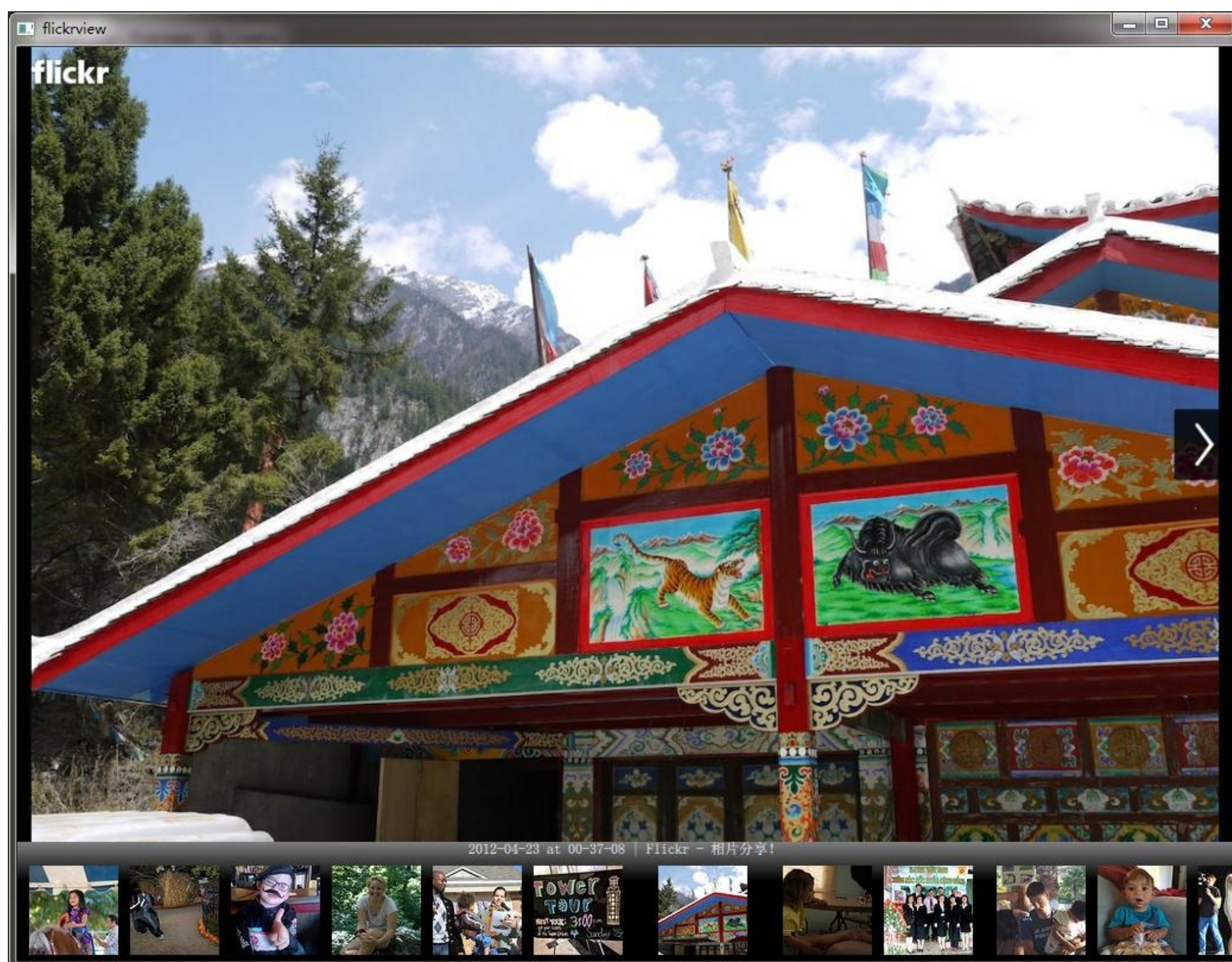
可以看到，欢迎界面和以前布局有了一些变动，更加清晰明了。但总体来说，整个界面及内容没有什么变化。

2 · 运行一个例子

我们点击"示例"，然后选择一个例子先来运行一下，比如这里选择Flickr View Example，这时会打开该程序并跳转到其帮助文档界面，如下图所示。



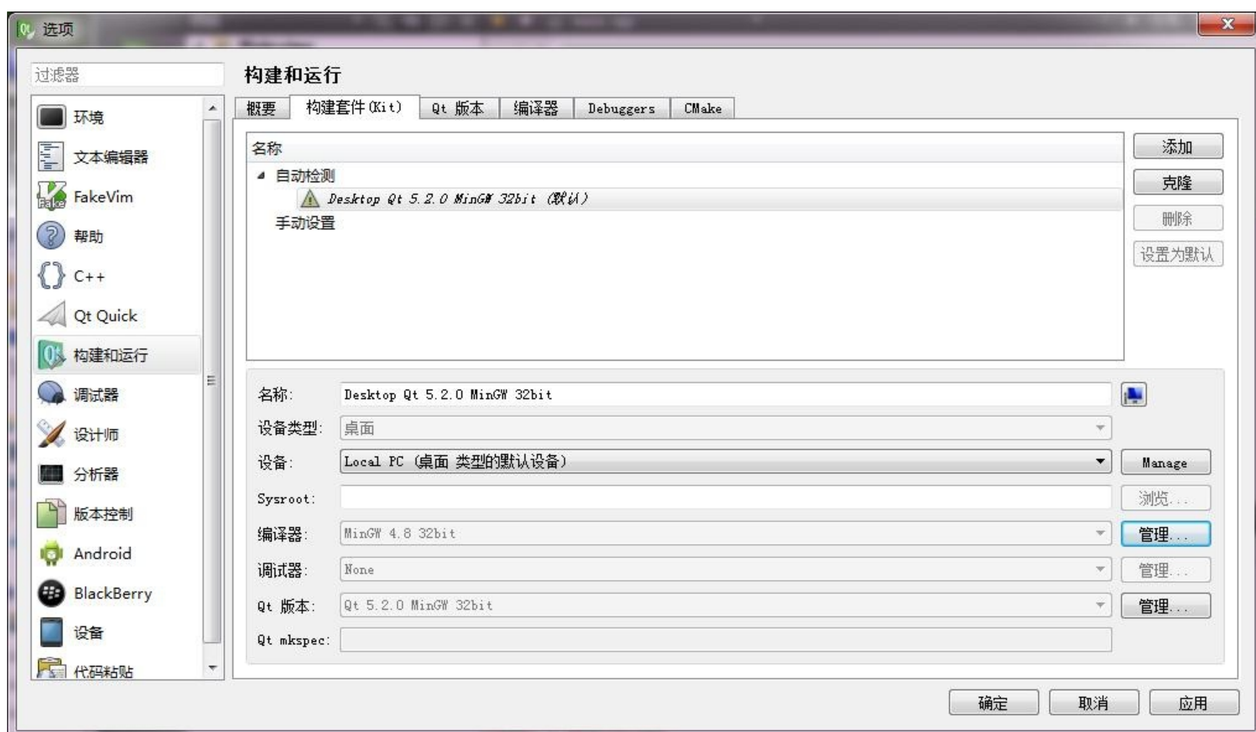
可以回到编辑模式简单看一下程序代码，然后点击运行按钮运行该程序，效果如下图所示。



这是个非常漂亮的图片浏览程序，是用qml编写的，不过这个并不是这里讲述的重点，非常先进且极具未来感的Qt Quick技术和QML语言会在后面专门的章节中详细讲解。这里要说的是，Qt 5已经是一个SDK了，它包含了开发所需要的大部分工具，包括了Qt Creator和MinGW，并做好了关联设置，所以我们可以看到，现在无需再像使用Qt 4.8那样手动设置就可以直接编译运行程序。

3．安装调试器

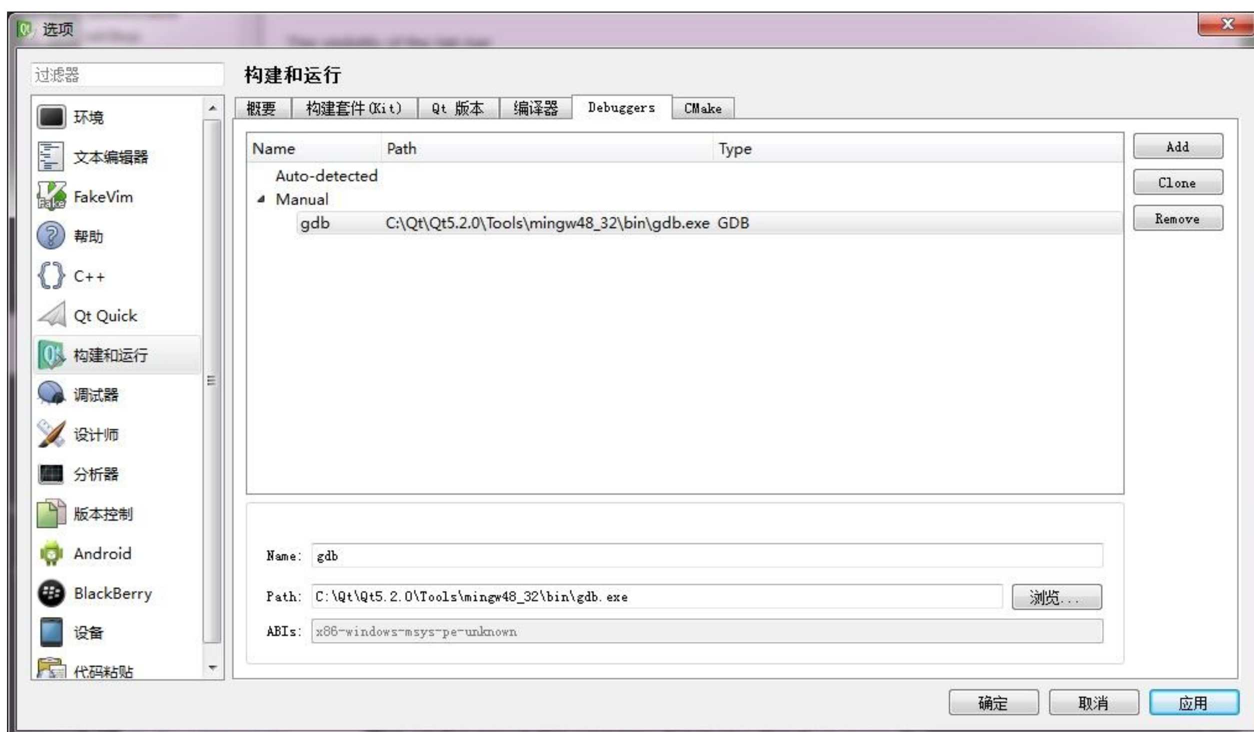
我们选择“工具”->“选项”菜单项，然后打开“构建和运行”页面中的构建套件，可以看到，这里已经自动检测到了一个构建套件。如下图所示。



不过，现在在构建套件前面有个黄色的感叹号，将光标移动到上面可以看到提示没有设置调试器。如下图所示。



在没有调试器的情况下，是无法启动调试模式的。这里，大家可以通过手动进行添加。先进入Debuggers标签页，可以看到现在这里还没有设置调试器，点击右侧的Add按钮，添加一个自定义的调试器，Name修改为 `gdb` ，Path选择Qt 5.2安装目录下的 `tool->mingw48_32->bin` 中的 `gdb` 程序，我这里是 `C:\Qt\Qt5.2.0\Tools\mingw48_32\bin\gdb.exe` ，完成后点击下面的应用按钮，效果如下图所示。



现在回到构建套件标签页，可以看到调试器已经默认选择为了我们添加的 `gdb` ，而且以前的黄色感叹号也消失了。

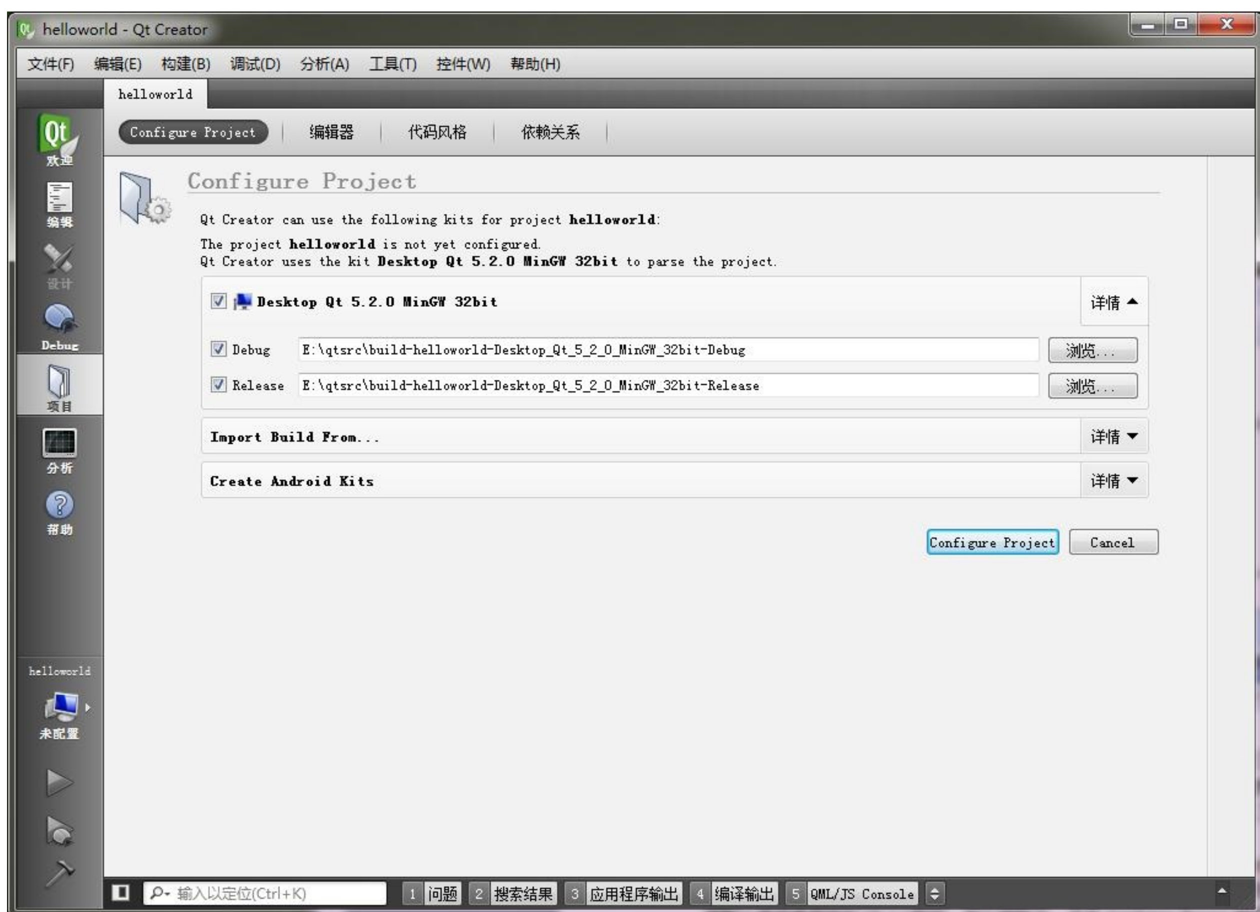
我们这里只是简单介绍了一下构建套件的设置，至于如何添加设置Android开发套件，会在后面专门的章节进行介绍，这里就不再讲解。

二、运行一个Qt 4程序

为了尽可能演示Qt 4程序在Qt 5编译时会出现的问题，我们这里使用了一个Windows Xp下面基于Qt 4.7创建的Qt Gui应用程序。这一节的目的就是让大家作为参考，如果你也遇到了类似的情况，那么可以这样来解决，如果没有遇到，则可以直接跳过相关内容。

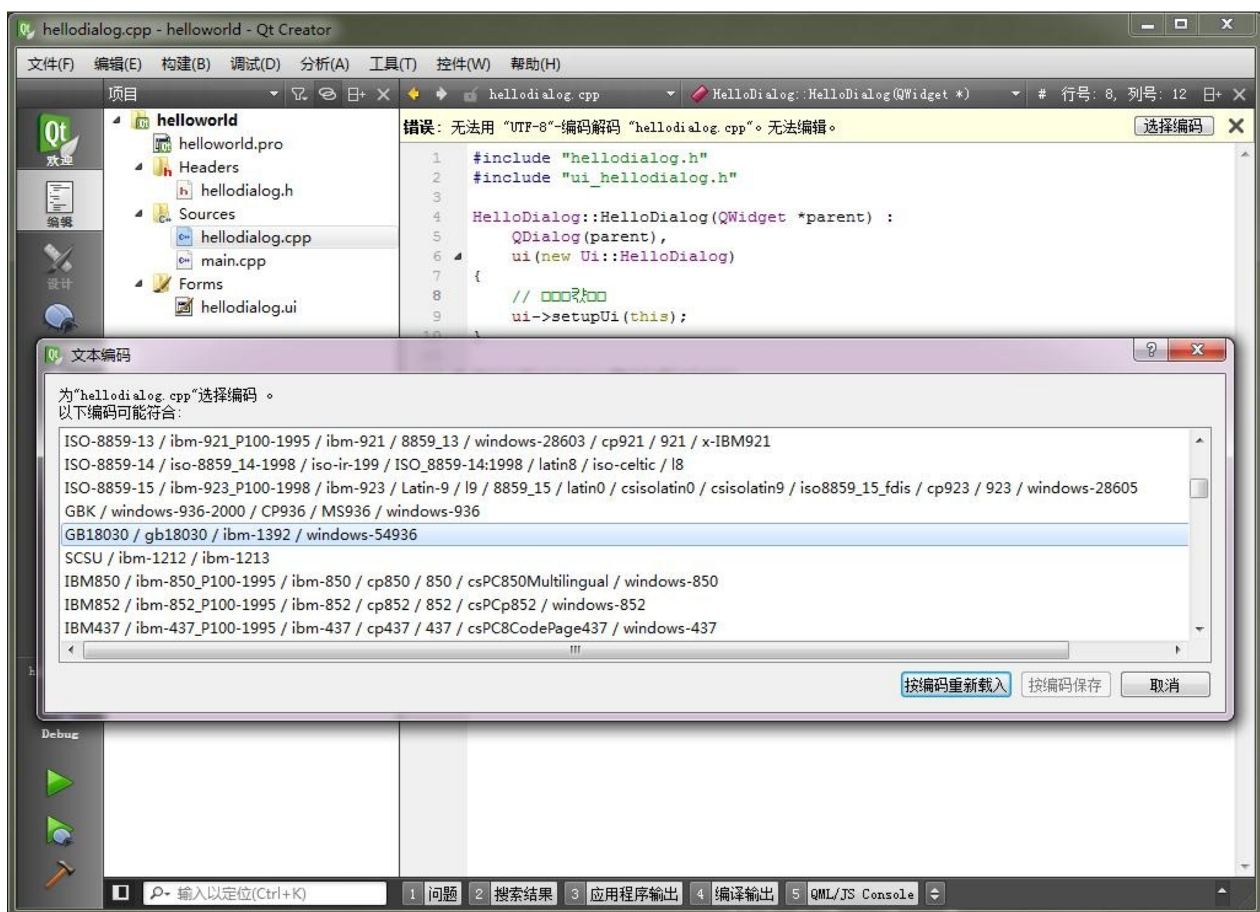
1 · 编码问题

我们打开现有的Qt 4版本的 `helloworld` 源码目录，然后将 `helloworld.pro` 文件拖入到Qt Creator中打开该项目，这时会跳转到项目模式，进行项目配置，也就是选择构建套件。这里默认使用桌面版的Qt 5.2即可，如下图所示，然后点击ConfigureProject按钮。

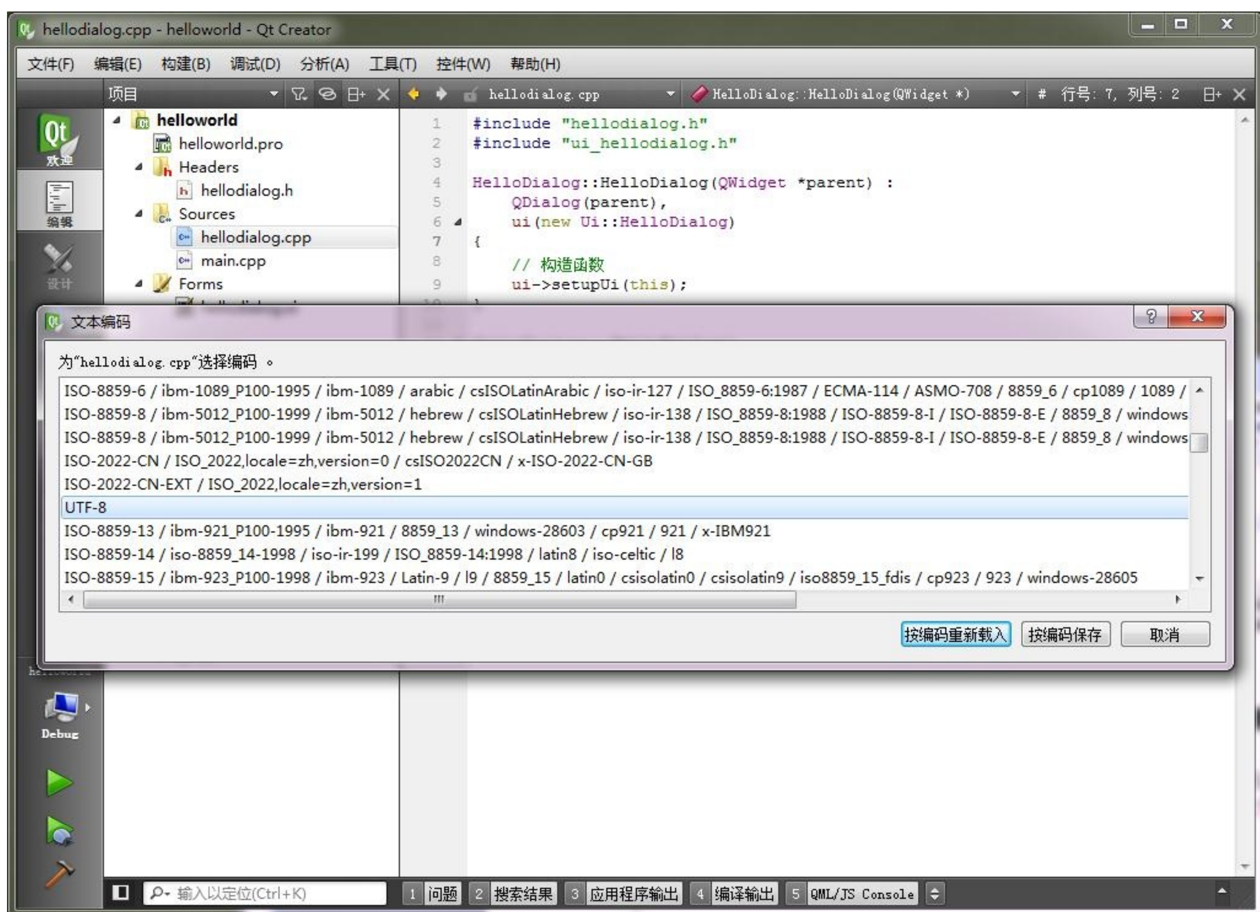


下面我们打开项目文件列表中的 `helloDialog.cpp` 文件，因为这里有一行中文注释，所以出现了“错误：无法用”UTF-8”-编码解码” `hellodialog.cpp` ”。无法编辑。”的错误提示，这是因为该文件不是使用UTF-8编码的，而其中的中文无法使用UTF-8自动解码造成的。为了使中文可以正常显示，并且以后不再出现该错误提示，我们可以通过下面的方法手动来将文件设置为UTF-8编码。

首先点击错误提示后面的选择编码按钮（也可以使用“编辑”->“选择编码”菜单项），然后选择 `GB18030/gb18030/ibm-1392/windows-54936` 一项，最后点击按编码重新载入按钮。如下图所示。



完成后发现已经可以正常显示中文了，但是如果关闭项目重新打开，中文依然无法正常显示。所以我们还需要继续设置。再次选择“编辑”->“选择编码”菜单项，然后选择UTF-8一项，点击按编码保存按钮。如下图所示。



这样设置完后，文件已经使用UTF-8进行保存了，后面再打开也不会出现编码错误了。

2．代码问题

下面先直接运行程序，这时会在问题面板出现 `QtGui/QApplication: No such file or directory` 的问题提示。如下图所示。



我们双击该问题，定位到出错位置，这时跳转到了 `main.cpp` 文件中，可以看到第一个头文件包含找不到路径。如下图所示。

```
main.cpp    main(int, c
1  #include <QtGui/QApplication>
2  #include "hellodialog.h"
3
4  int main(int argc, char *argv[])
5  {
6      QApplication a(argc, argv);
```

我们可以到QtGui目录（我这里是：`C:\Qt\Qt5.2.0\5.2.0-beta1\mingw48_32\include\QtGui`）中查看一下，发现这里没有`QApplication`相关文件。为了更加明了和准确的讲解该问题，我们在Qt Creator中创建一个基于Qt 5.2的GUI程序作为参照。

选择“新建”->“新建文件或项目”菜单项，这里可以看到在应用程序中第一个是QtWidgets Application，而不再是Qt 4中熟悉的Qt GuiApplication，我们选择它作为模板。然后添加项目名称为`helloqt`，路径大家选择一个没有中文的目录即可。下面的Kit就选择默认的DesktopQt 5.2，然后类信息不用更改。

完成之后，我们先运行一下新建的`helloqt`程序，发现是没有问题的。这时打开其`main.cpp`文件，发现`#include <QApplication>`是这样写的，这里没有添加QtGui。现在我们更改前面`helloworld`项目中`main.cpp`文件的头文件包含为`#include <QApplication>`，不过，改成这样后依然提示找不到文件。

现在我们可以对照`helloqt`文件的内容，看看还有哪里与我们Qt 4程序不同。这时，在`helloqt.pro`文件中会很明显发现一行代码：

```
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

这行代码的大致意思是：在高于Qt4的版本中要添加`QT += widgets`，也就是说要使用`widgets`模块，这里的`widgets`模块到底包含了什么内容，有什么作用？这些问题我们暂且不考虑，现在将这行代码复制到`helloworld.pro`中，然后运行`helloworld`程序，发现程序已经可以可以正常运行了。

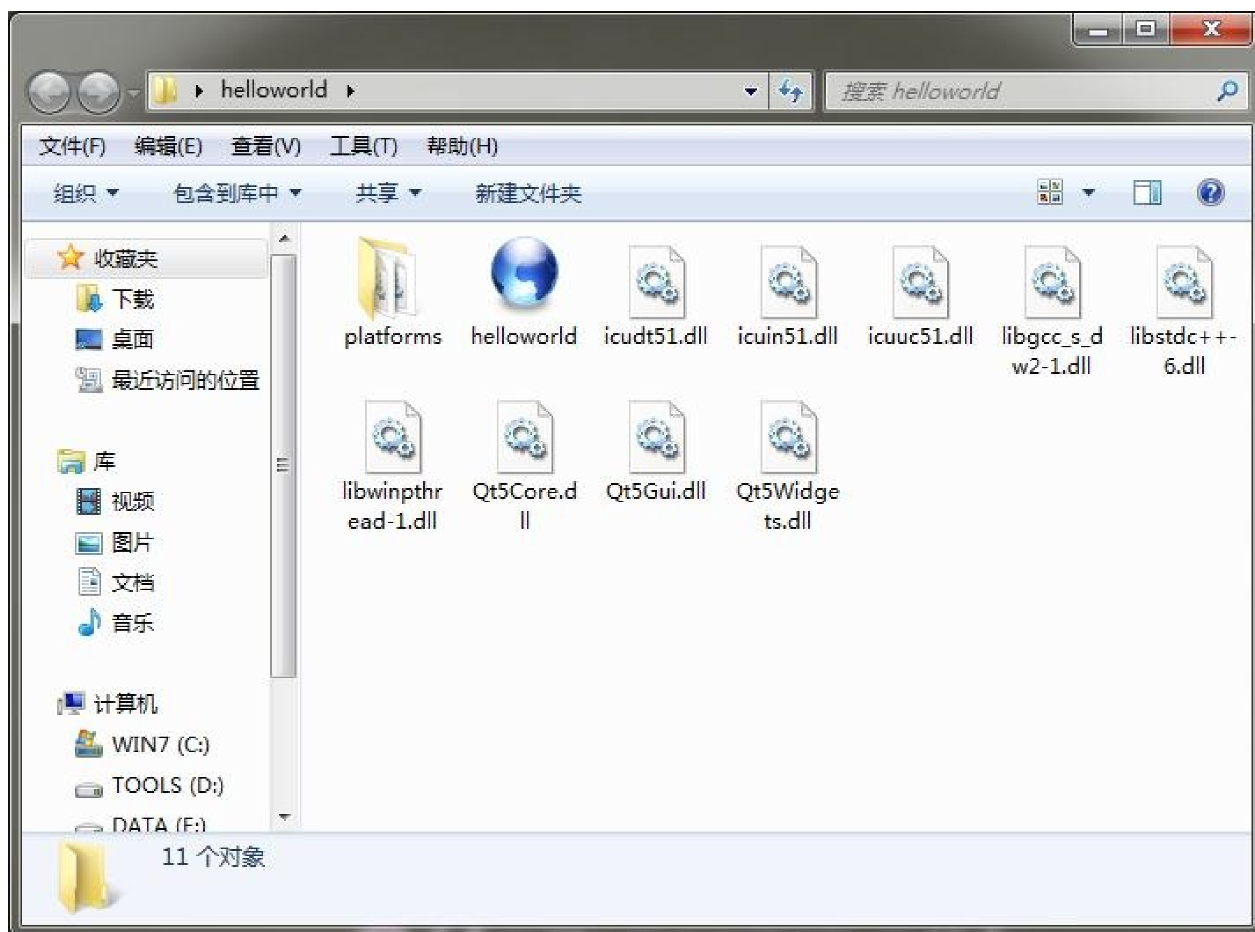
3．应用程序图标

在这一节的最后，我们再补充一点。在Qt 4中如果要给一个程序添加应用程序图标，需要先有一个`ico`图标文件，然后创建一个`.rc`文件，还要输入一行怪异的代码。而在Qt 5中这个变得非常简单，我们只需要将`ico`图标文件放到源码目录，然后在`pro`项目文件中添加一行代码`RC_ICONS = myico.ico`即可，后面`myico.ico`就是自己图标文件的名字。

三、发布Qt 5程序

现在`helloworld`程序已经可以编译运行了，下面我们将打包发布该程序。要作为发布使用，先要选择编译Release版本，然后运行。完成后到编译生成目录（我这里是：`E:\qtsrc\build-helloworld-Desktop_Qt_5_2_0_MinGW_32bit-Release\release`）中将生成的`helloworld.exe`文件复制到一个新建的文件夹中，比如这里放到了新建的`helloworld`文件夹中。然后双击运行`helloworld`程序，并根据提示到Qt 5.2的安装目录（我这里

是：`C:\Qt\Qt5.2.0\5.2.0-beta1\mingw48_32\bin`）中将需要的dll文件复制过来，一共是9个。这样就可以在本机上运行该程序了，但是在别的没有安装该版本Qt的机子上还是无法运行，这时需要将`C:\Qt\Qt5.2.0\5.2.0-beta1\mingw48_32\plugins`中的`platforms`目录复制过来，而里面只要保留`qminimal.dll`和`qwindows.dll`两个文件即可。最终效果如下图所示。



后面就可以将该文件夹通过压缩文件打包进行发布了。当然，如果程序中使用了其他模块，可能还需要复制`plugins`目录中的相应的文件。

结语

对于大部分Qt 4程序而言，Qt 5没有太大的改变，不过在升级移植的过程中还是会发现很多细节改动的。这一节我们讲述了Qt 5.2版的安装、设置，然后讲述了怎样将一个Qt 4程序使用Qt 5进行编译运行，最后还讲述了Qt 5程序的发布。

在下一节我们将会讲解Qt 5的整个框架，让大家更加清楚Qt 5中改变了哪些模块，增加和删除了哪些模块。

涉及到的源码:

- helloqt.zip
- helloworld.zip

从Qt 4到Qt 5（二）Qt 5框架介绍

导语

上一节已经安装好Qt 5.2，并将一个Qt 4程序迁移到了Qt 5上。其中我们讲到Qt 5中 `QApplication` 类已经不在QtGui模块中了，而且所有的Qt 5图形界面程序都必须在 `.pro` 项目文件中添加 `widgets` 模块。那么到底Qt 5中对模块进行了哪些改动，Qt 5的框架又是怎样的？这一节将和大家一起看一下这些内容。

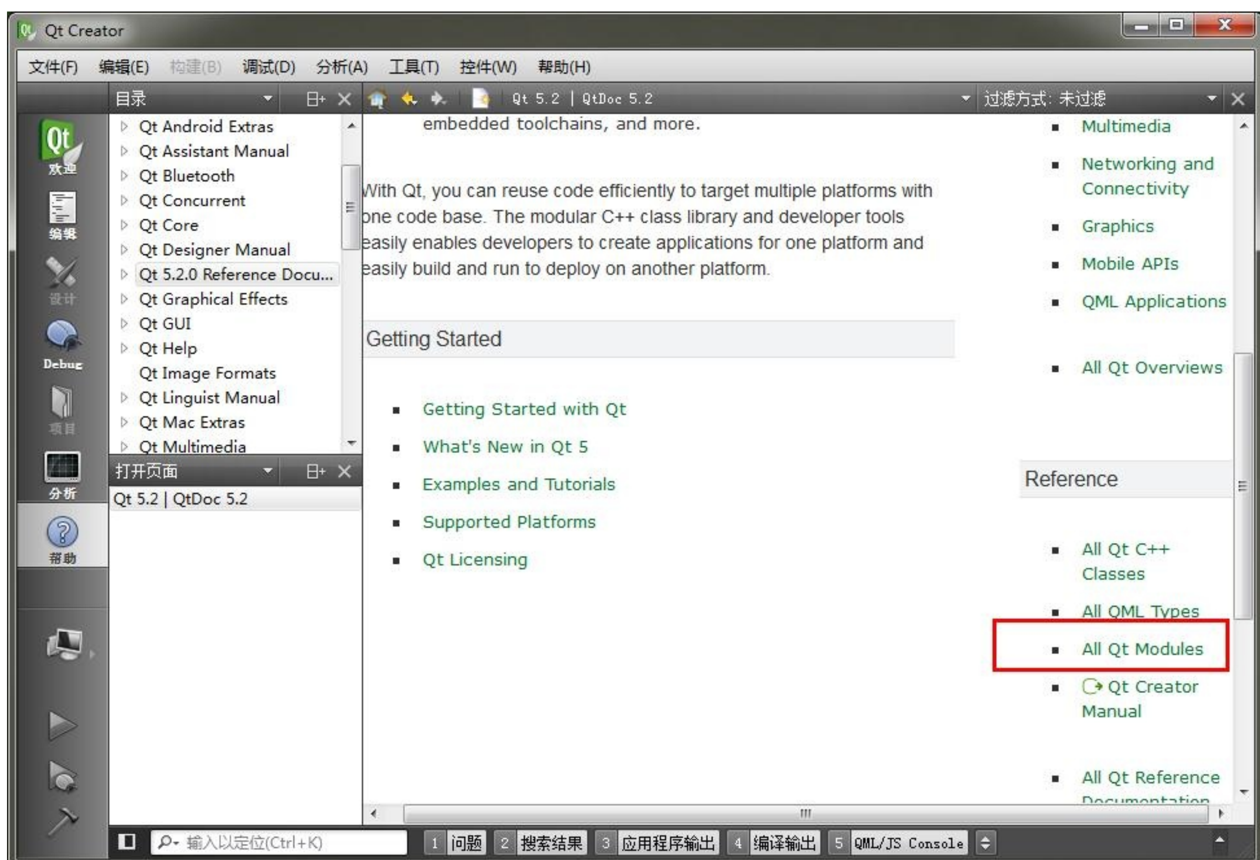
环境：Windows 7 + Qt 5.2.0+QtCreator 3.0

目录

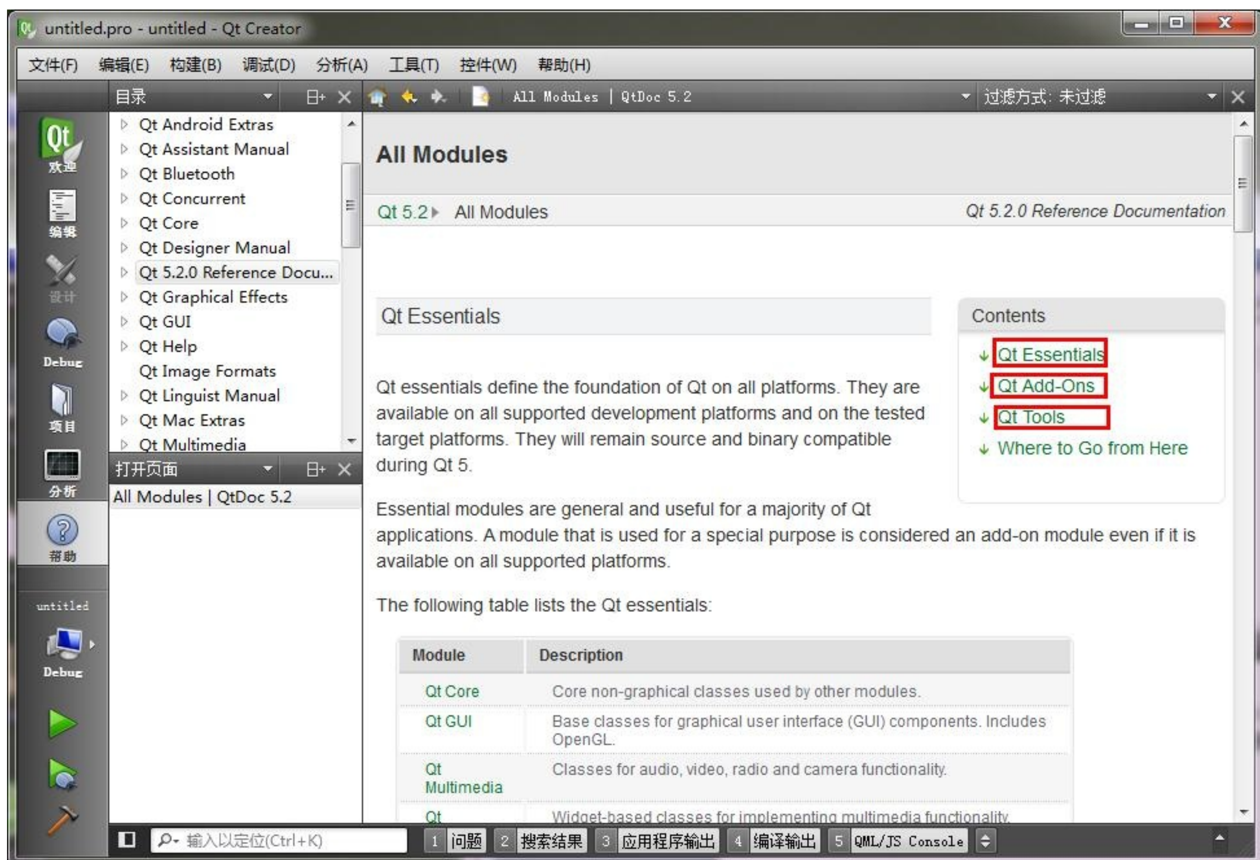
正文

一、在帮助中查看所有模块

打开Qt Creator，进入其帮助模式，然后选择目录方式进行查看，打开“Qt 5.2.0ReferenceDocumentation”页面。在这里提供了Qt5.2的整体介绍，并将其所有内容进行了分类。我们选择右下角的“All Qt Modules”来查看所有的Qt模块。如下图所示。

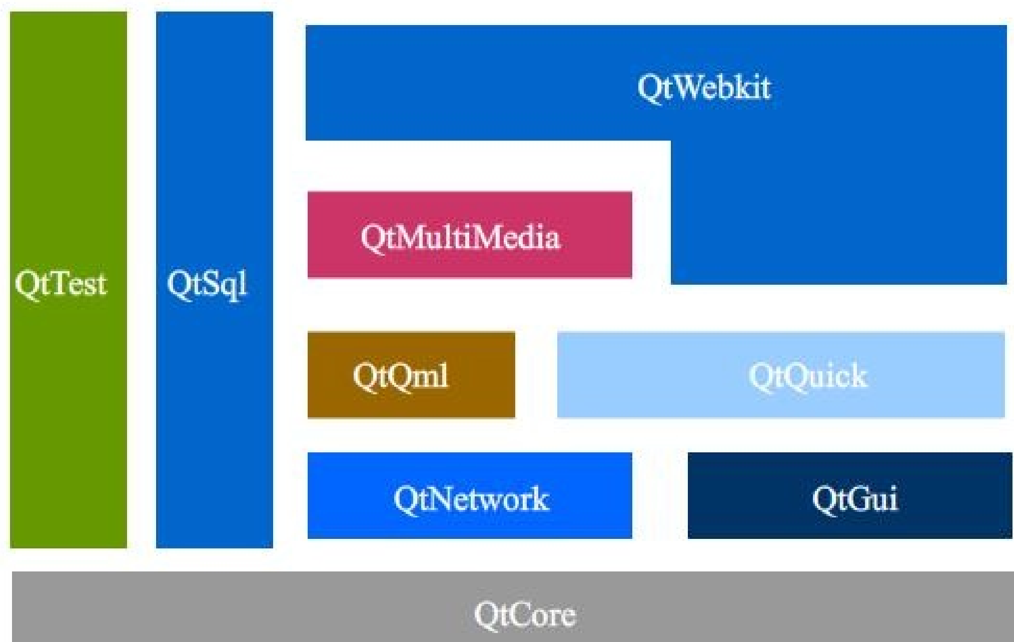


在所有模块页面，将Qt的模块分为了三部分：Qt 基本模块（Qt Essentials）、Qt扩展模块（Qt Add-Ons）和Qt工具（Qt Tools）。Qt基本模块中包含了Qt核心基础的功能，这个我们会在后面详细讲解；而Qt扩展模块包含了以前 `QtMobility` 中的一些与移动有关的模块，如蓝牙 `QtBluetooth` 、传感器 `QtSensors` 等。还包含了以前Qt 4中的一些模块，例如 `QtDBus` 、 `QtXML` 、 `QtScript` 等。除此之外，还新添了一些模块，例如图形效果 `QtGraphicalEffects` 、串口 `Qt Serial Por` t、还有出现在商业版中的 `Qt3D` 等。这些模块都是有特殊用途的，它们很多需要在特殊的平台上才可使用。在扩展模块中我们也看到了 `Qt Print Support` 打印支持模块，它是以前很多类的重组模块；在Qt工具中包含了Qt设计器、Qt帮助和Qt界面工具三部分内容。如下图所示。



二、Qt基本模块框架

Qt基本模块中定义了适用于所有平台的Qt基础功能，在大多数Qt应用程序中需要使用该模块中的提供的功能。Qt基本模块的底层是 `QtCore` 模块，其他所有模块都依赖于该模块，这也是为什么我们总可以在 `.pro` 文件中看到 `QT += core` 的原因了。整个基本模块的框架如下图所示。

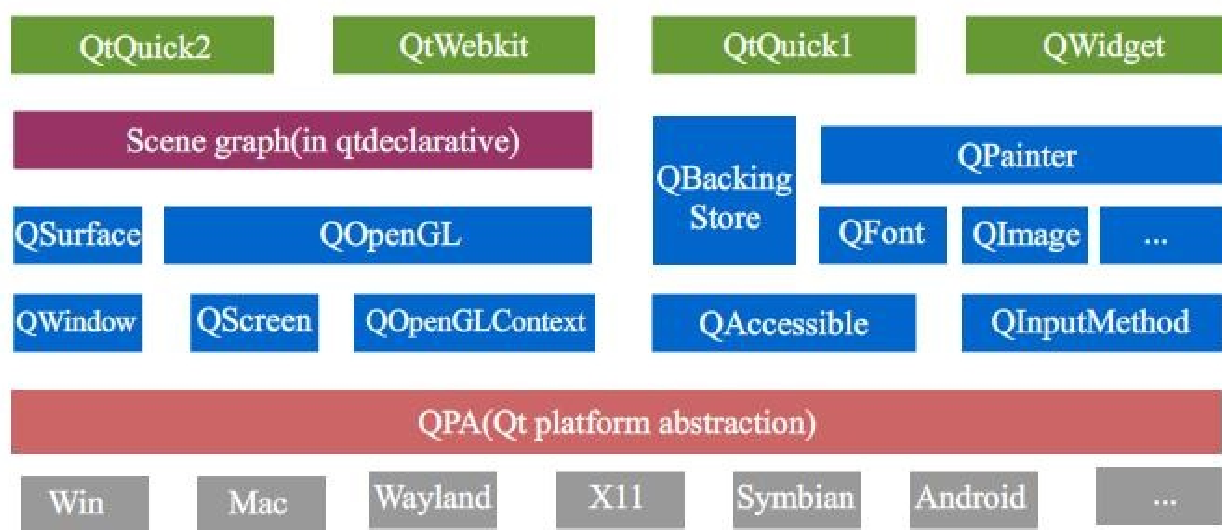


最底层的是 `QtCore` ，它提供了元对象系统、对象树、信号槽、线程、输入输出、资源系统、容器、动画框架、JSON支持、状态机框架、插件系统、事件系统等等所有基础功能。该模块的重要性不言而喻。在其之上，直接依赖于 `QtCore` 的是 `QtTest` 、 `QtSql` 、 `QtNetwork` 和 `QtGui` 四个模块，其中测试模块 `QtTest` 和数据库模块 `QtSql` 是相对独立的，而更加重要的是网络模块 `QtNetwork` 和图形模块 `QtGui` ，在它们两个之上便是Qt 5的重要更新部分 `QtQml` 和 `QtQuick` 。而最上层的是新添加的 `QtMultiMedia` 多媒体模块，和在其之上的 `QtWebKit` 模块。

对于整个框架，大家可以理解为下层模块为上层模块提供支持，或者说上层模块包含下层模块的功能。举个例子，例如 `QtWebKit` 模块，它既有图形界面部件也支持网络功能，还支持多媒体应用。对于其他模块，我们这里就不再深入介绍，下面主要来讲解一下其中最重要的QtGui模块。

三、图形界面库框架

现在再回到开头的问题，我们已经发现 `QApplication` 不在 `QtGui` 模块中了，其实不仅如此，就连所有用户界面的基类 `QWidget` 也不在 `QtGui` 模块中了，它们被重新组合到了一个新的模块 `QtWidgets` 中。Qt 5的一个重大更改就是重新定义了QtGui模块，它不再是一个大而全的图形界面类库，而是为GUI图形用户界面组件提供基类，包括了窗口系统集成、事件处理、OpenGL和OpenGL ES集成、2D绘图、基本图像、字体和文本等内容。在Qt 5中将以前 `QtGui` 模块中的图形部件类移动到了 `QtWidgets` 模块中，将打印相关类移动到了 `Qt Print Support` 模块中。不过Qt 5中去掉了 `QtOpenGL` 模块，而将OpenGL相关类移动到了 `QtGui` 模块中。有的读者可能发现在Qt扩展模块中依然有 `QtOpenGL` 模块，其实它只是为了便于Qt 4向Qt 5移植才保留的，在编写Qt 5程序时依然强烈推荐使用 `QtGui` 模块中的 `OpenGL` 类。了解了图形库的大体更改，下面我们来看一下Qt图形界面库的整体框架。如下图所示。



在各种支持的平台之上是底层的平台抽象层QPA，这个就是被称作LightHouse的灯塔项目，它是Qt可以无处不在的基础。而在其上的所有蓝色组块都是 `QtGui` 模块的内容，它们被分为了两类，一类以OpenGL为核心，它是现在最新的 `QtQuick2` 和 `QtWebkit` 的基础；一类是以辅

助访问和输入方式为基础的一般图形显示类，它们是经典 `QWidget` 部件类和 `QtQuick1` 的基础。

四、 `QtQml` 和 `QtQuick` 框架

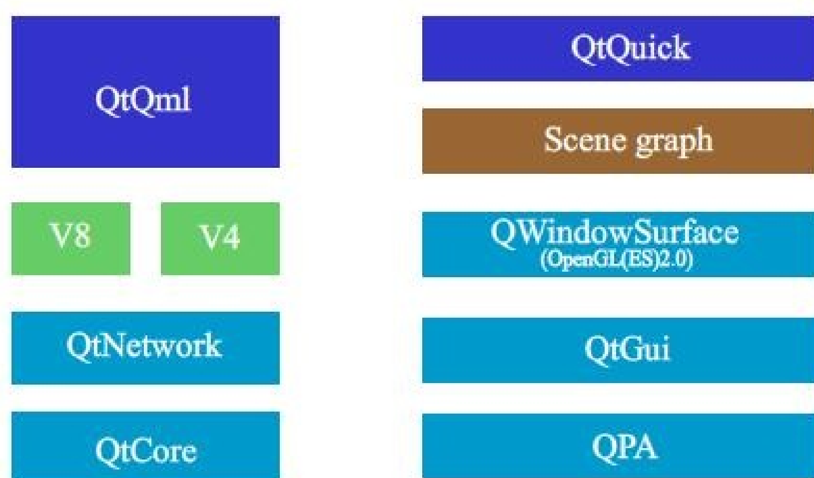
如果要问Qt 5最大的更新和特色是什么，那非 `QtQml` 和 `QtQuick` 莫属。其实，在Qt 4.7的时候就已经有 `QtQuick` 了，不过它在那个时候并不成熟。只有到现在的Qt 5，qml和quick才发展壮大，逐渐规范起来，并且拥有了与 `QWidget` 平分秋色的地位。大家可能已经了解到，qml和quick是为移动设备而生的，并且是今后Qt发展的方向。

`QtQuick` 在Qt 4中是这样定义的：Qt Quick 是一种高级用户界面技术，使用它可轻松地创建供移动和嵌入式设备使用的动态触摸式界面和轻量级应用程序。三种全新的技术共同构成了 `Qt Quick` 用户界面创建工具包：一个改进的Qt Creator IDE、一种新增的简便易学的语言(QML) 和一个新加入 Qt 库中名为 `QtDeclarative` 的模块，这些使得 Qt 更加便于不熟悉 C++ 的开发人员和设计人员使用。

不过在Qt 5中将以前的 `QtQuick` 分为了两大部分： 一部分是 `QtQml` ，它提供了一个QML语言框架，定义并实现了语言引擎基础，还提供了便于开发者使用的API，实现使用自定义类型来扩展QML语言以及将JavaScript和C++集成到QML代码中。另一部分是新的 `QtQuick` ，它是一个用于编写QML程序的标准库，它提供了使用QML创建用户界面程序时需要的所有基本类型。

在Qt 5中已经很明了地分离了qml和quick，使得我们可以对这项新技术拥有更加清楚地认识。与其说这是一项新技术，不如说这是Qt创造的一个新的语言和类库，请允许在这里打个不太科学的比方：qml就好比是C++语言，那么quick就是Qt库，Qt库是用C++语言编写的一个类库，而quick就是用qml语言编写的一个类库，只不过在qml的世界里，没有类这个叫法而已。

`QtQml` 和 `QtQuick` 的框架如下图所示。



可以看到 `QtQml` 和 `QtQuick` 是独立的两部分： `QtQml` 以 `QtCore` 为基础，拥有 `QtNetwork` 的相关功能，然后搭建在了V8和V4两个JavaScript引擎上，V8大家应该已经熟知了，而V4是一个轻量级的JavaScript引擎。不过这里需要提及一下，在最新的Qt 5.2版本中，V8已经完全被

一个新的Qt专有引擎代替了，原因是V8适用于浏览器却不太适用于qml。我们也可以看到 `QtQml` 本身并没有涉及图形显示的内容； `QtQuick` 以 `QPA` 为基础，而后经过了 `QtGui` 、OpenGL和Scene graph三层封装，这里可以看到，新的 `QtQuick` 是建立在OpenGL之上的，并且使用了新的Scene graph进行图形渲染。很明显， `QtQuick` 就是用于图形显示的。

结语

从Qt 4到Qt 5，整个框架进行了优化调整，目的就是为了达到更好的性能和以后进一步地扩展。可以发现，OpenGL和WebKit在整个框架中占有举足轻重的地位，不过在不远将来的Qt 5.3，Chromium将代替WebKit成为Qt的Web引擎，因为Chromium提供了更好的跨平台性和其他一些易用性。

入门篇

第51篇 Qt 5.5全新的开始

导语

时间转眼而逝，看一下上次发的教程，已经是一年前的事情了。这一年发生了很多事情，包括自己也包括Qt。当然，自己很忙或者说为了编写《Qt 5编程入门》这些理由，并不能为一年的搁置进行开脱，所以这里首先还是要向广大读者，跟随yafeilinux一起走来的朋友说声抱歉，让你们久等了。

我一直把写博文写教程当做是一种爱好，即便是技术类博文也是如此，想到哪里就写哪里，少了点技术类文章的严谨，多了点抒情类文章的随性。这也是我教程的一种风格，我坚信只有爱好的东西才能做到完美，做大极致，编程亦如此。一年没有更新博文，也是觉得有些时候，有点急功近利了，这不符合我写这个系列教程的初衷，所以即便很多朋友邀我尽快更新，我还是没有为了应付而进行大幅度更新。

其实这一年中我也一直在做一些和Qt有关的事情，比如说前面提到的《Qt 5编程入门》，这个是和豆子（devbean）一起写的，现在已经出版上市了。在写这本书的同时，想了很多，也有很多好的东西想和大家分享，但是还是因为精力有限......再比如说为《Qt Creator快速入门》编写了实验讲义和PPT课件，完成这个的时候，我对该书第三版已经有了大致的思路，也本想早早和大家探讨，但还是因为精力有限......再有就是和天嵌科技合作的嵌入式教程，这个也是和hzzhou合作的教程，年初的时候用三个月跟hzzhou合作开发了一个小项目，这次再次合作，将探索Linux嵌入式编程教程的编写。

好了，好像扯了很多废话，现在终于有时间让自己静下心来了，万事开头难，只要开始了就要坚持下去，以后的一段时间，将以更新教程为主要业余工作。这一篇之所以叫全新的开始，既是因为Qt开源（qt-project.org）和Qt商业（qt.digia.com）进行合并，成立了Qt全资子公司，而且发布了全新的qt.io网站，也是因为最新的Qt 5.5版本已经公布，还有就是，一年没有写网络教程了，风格和思路可能会与以前有所不同。 作为全新开始的第一篇，这里不涉及太多的技术问题，而是讲一些新接触Qt应该了解的内容。后面的章节我们会从Qt Quick编程讲起，如果要学习C++ Widget编程，可以参考前面的文章。

环境：Windows 7 + Qt 5.5.0+QtCreator 3.4.2

目录

- 五、运行一个示例程序

正文

一、下载Qt

这里先说一下，很多同学不知道Qt去哪里下载，尤其是老版本的Qt根本不知道哪里可以找到，其实现在可以到 http://download.qt.io 上下载所有Qt开源的内容，其目录如下图所示。

Name	Last modified	Size	Metadata
snapshots/	26-Mar-2014 11:52	-	
online/	13-Mar-2014 08:45	-	
official_releases/	17-Dec-2014 09:57	-	
ministro/	07-May-2013 17:46	-	
learning/	22-May-2013 16:20	-	
development_releases/	25-Sep-2014 14:31	-	
community_releases/	31-Jan-2014 09:29	-	
archive/	16-Dec-2014 10:39	-	

这里最主要的目录是 `official_releases` ，其中提供了官方发布的正式版软件。就是说要使用正式发布的稳定版Qt、Qt Creator等，到这里下载；而在archive目录中是存档内容，这里是Qt和Qt Creator的仓库，里面包含了所有的版本，所以如果想找老版本的同学可以在这个目录中查找。其他几个目录提供了一些相关的软件或者工具，比如在snapshots目录中可以下载到最新版本的快照，不过可能只提供了源码还需要自己进行编译。

这里我们下载 `development_releases→qt→5.5→5.5.0` 中的：

```
qt-opensource-windows-x86-android-5.5.0.exe
```

为了便于大家下载，在我们qter论坛的下载页面提供了便捷下载链接，如下图所示。

Home → **下载**

版权声明：从本社区下载的所有资源不得用于商业用途！

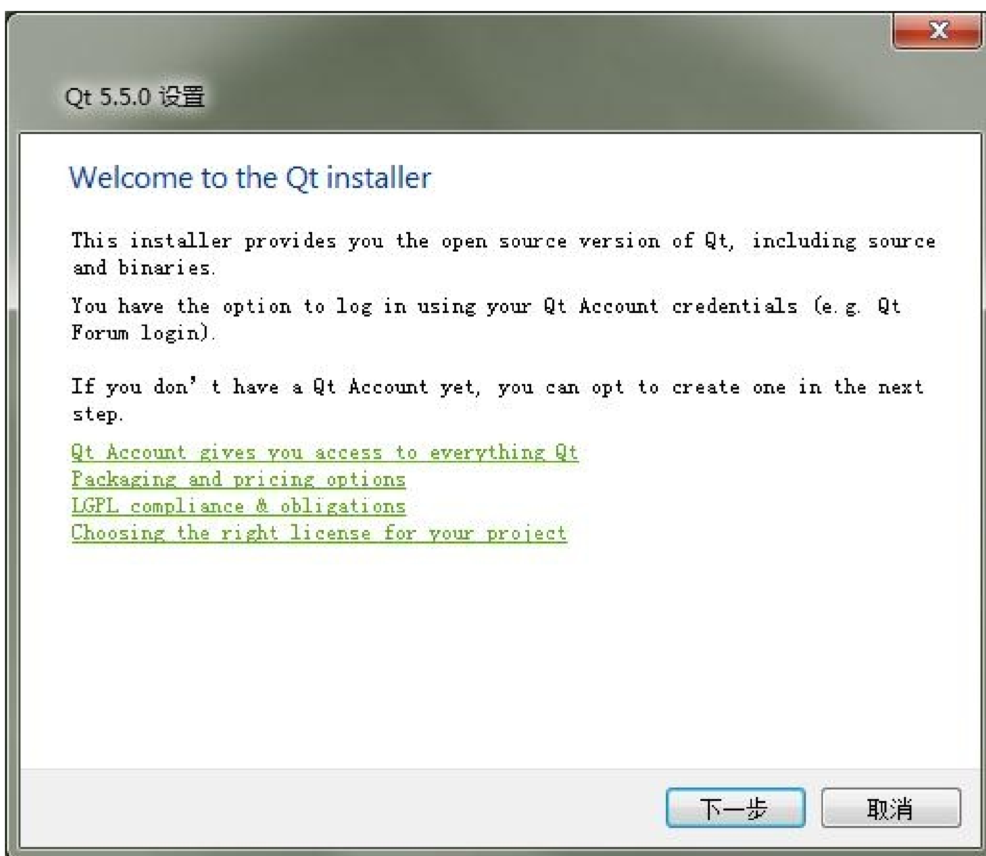


比如点击“Qt SDK下载”按钮进入页面，进入后会默认显示最新Qt版本的下载，如果要下载其他内容，可以点击Parent Directory跳转到上一层目录。

Name	Last modified	Size	Metadata
Parent Directory		-	
5.5/	01-Jul-2015 09:13	-	
5.4/	02-Jun-2015 07:53	-	
5.3/	16-Sep-2014 08:45	-	
5.2/	24-Feb-2014 13:07	-	
5.1/	06-May-2014 12:44	-	
5.0/	03-Jul-2013 11:57	-	
4.8/	25-May-2015 17:55	-	

二、安装Qt 5.5

进行Qt 5.5的安装，与以前的版本稍微不同的地方是，一开始会提示让登陆或者注册Qt社区账户。如下图所示。



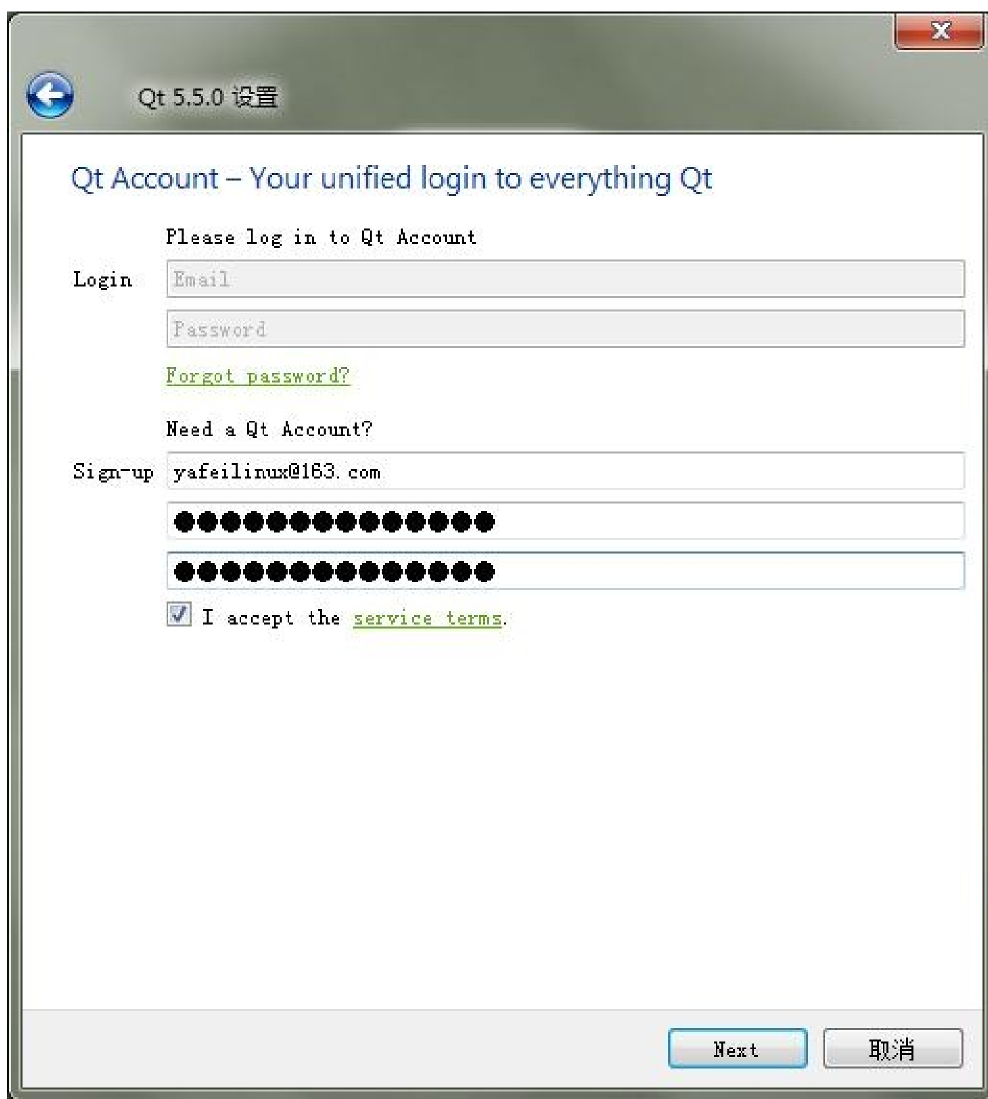
如果已经有了一个Qt账户，可以直接填写Login后面的账户邮箱和密码；如果没有注册过，可以在Sign-up后面填写要注册的邮箱、密码。如下图所示。



当然也可以点击Skip按钮直接跳过这一步。不过拥有一个Qt账户还是很有用的，比如可以到Qt论坛进行发帖等。所以我们这里选择注册一个新的Qt账户，这里要注意，输入的密码至少要7位，不能包含上面输入的邮箱地址，而且要包括小写字母、大写字母、数字和符号四种类型中的三类。如下图所示。



输入完密码后，勾选下面的同意服务条款选项，然后就可以点击Next按钮继续安装Qt了。如下图所示。



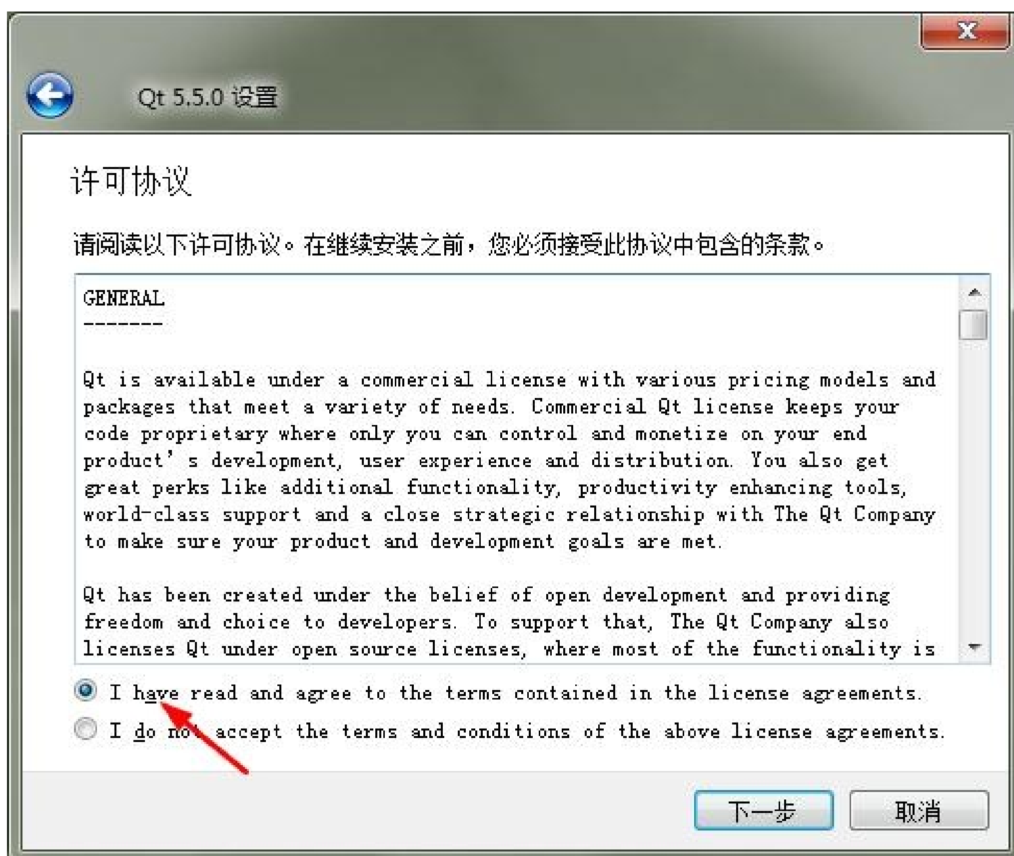
首先要设置安装路径，注意安装路径中不要有中文。这里我们选择默认的路径，如下图所示。



下一步是选择组件，默认的选项已经满足我们的开发要求了，如果有其他要求，比如需要源码组件或者Android armv5开发，可以进行相关选择。如下图所示。



再往下是勾选同意许可协议。如下图所示。

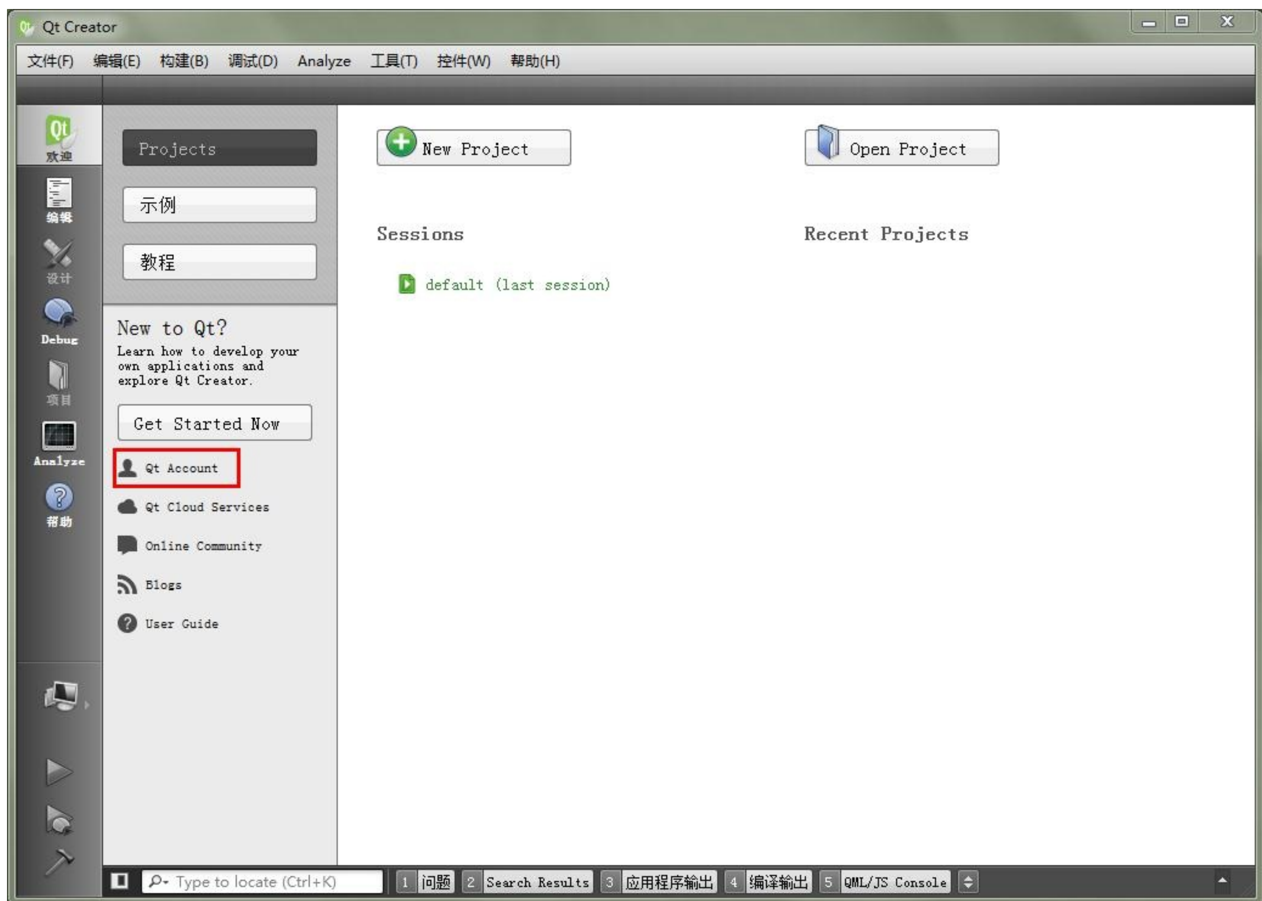


当安装完成后，默认Launch Qt Creator是勾选的，这样点击完成按钮就会自动启动Qt Creator，如下图所示。



三、激活Qt账户

QtCreator运行后如下图所示，还是我们熟悉的界面。在正式讲解前，我们再来说说Qt账户，点击欢迎界面的QtAccount链接，可以快捷登陆Qt账户。



这时会弹出浏览器窗口，显示Qt账户登陆界面。我们可以输入前面注册过的账户，然后点击Sign in按钮，如下图所示。

Qt Account

Your unified login to everything Qt

yafeilinux@163.com

Sign in

Forgot password?

Create Qt Account

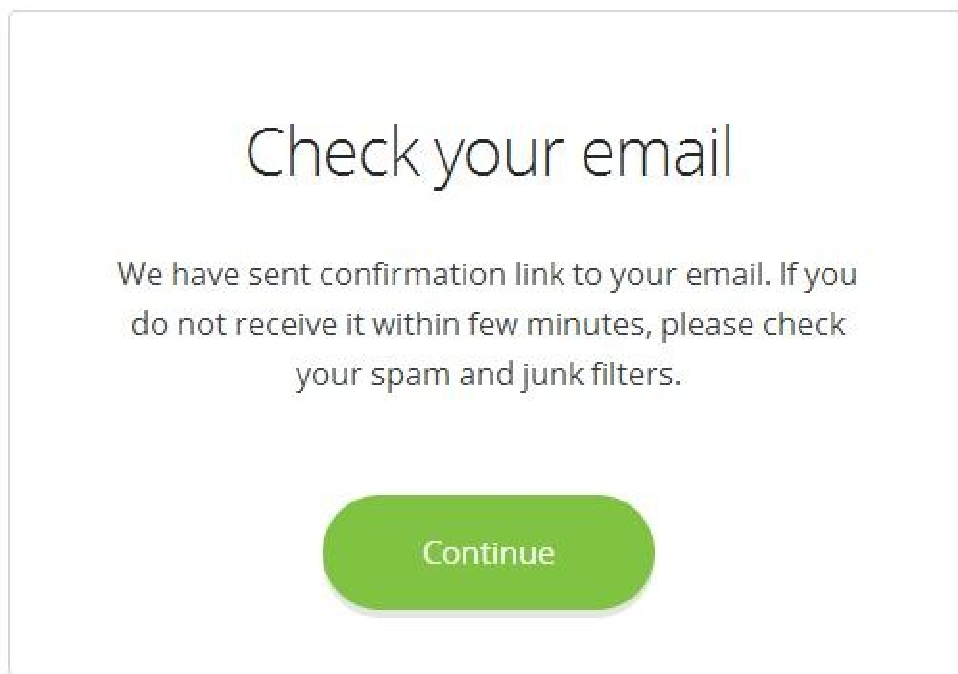
如果是第一次登陆，会提示让验证邮箱，点击如下图所示的链接。

Your unified login to everything Qt

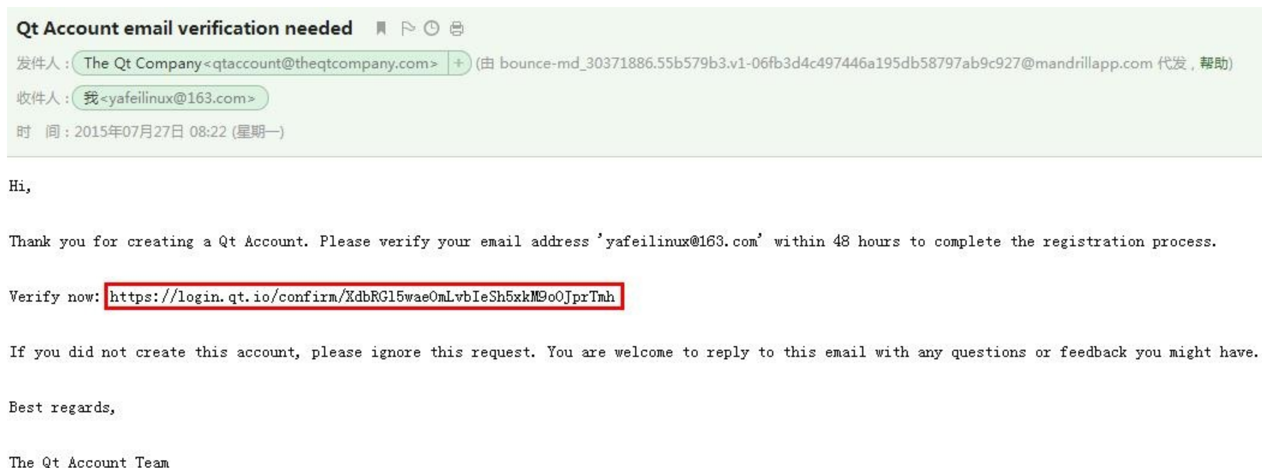
Your email address has not yet been verified

Confirm your email to get full functionality. Check your email for further instructions.
Resend the instructions to your email.

这时会提示已经向注册的邮箱发送了验证链接，如下图所示。



现在我们登陆自己注册的邮箱，打开Qt发送来的邮件，然后将Verify now后面的链接复制粘贴到浏览器地址栏进行访问。如下图所示。



Qt Account email verification needed

发件人：The Qt Company<qtaccount@theqtcompany.com> (由 bounce-md_30371886.55b579b3.v1-06fb3d4c497446a195db58797ab9c927@mandrillapp.com 代发，帮助)

收件人：我<yafeilinux@163.com>

时 间：2015年07月27日 08:22 (星期一)

Hi,

Thank you for creating a Qt Account. Please verify your email address 'yafeilinux@163.com' within 48 hours to complete the registration process.

Verify now: https://login.qt.io/confirm/XdbRGl5waeOmLvbIeSh5xkM9oOJprTmh

If you did not create this account, please ignore this request. You are welcome to reply to this email with any questions or feedback you might have.

Best regards,

The Qt Account Team

在弹出的页面填写一些个人信息后，点击Confirm按钮，如下图所示。

Qt

Your Account

yafei

huo

Qt

Home

China

Beijing

Phone

Send me news about Qt

Confirm

现在就完成了Qt账户的验证，登陆Qt账户以后，可以更好的使用Qt社区的一些内容，比如Documentation文档、Blog博客、WiKi百科、Forum论坛等。如下图所示。

Qt Account yafei huo

Qt Account Home
My Profile
Support Center



Your unified login to everything Qt

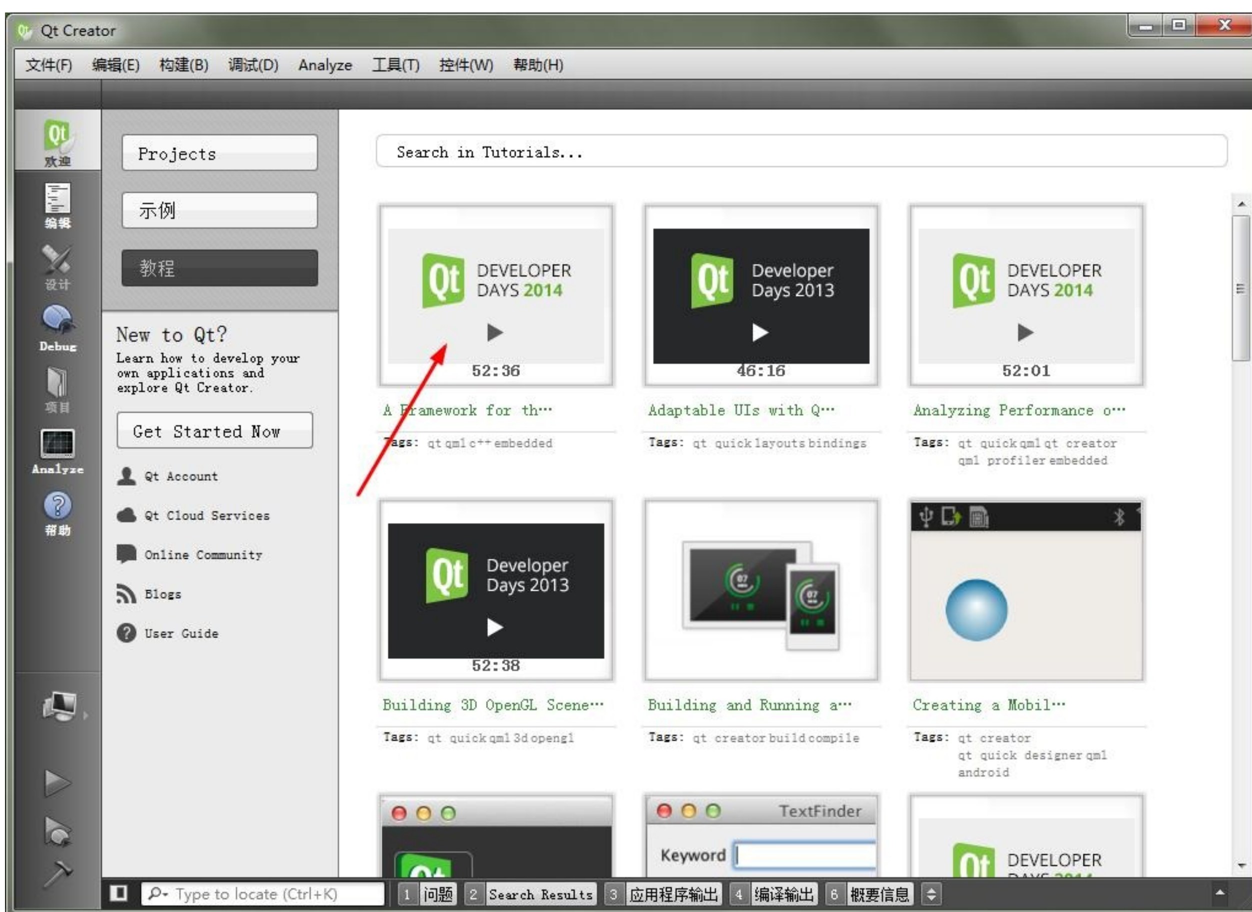
Exclusive offer for Qt Account users:
› The Qt Company Support FAQ knowledge base

You don't seem to have a Qt commercial license. Buy Qt Now or Start for Free.

Useful Quick Links:
› Documentation
› Getting Started
› Qt Blog
› Qt Wiki
› Qt Forum

四、学习Qt视频教程

在欢迎界面点击“教程”按钮，可以看到Qt提供的一些视频教程，如下图所示。



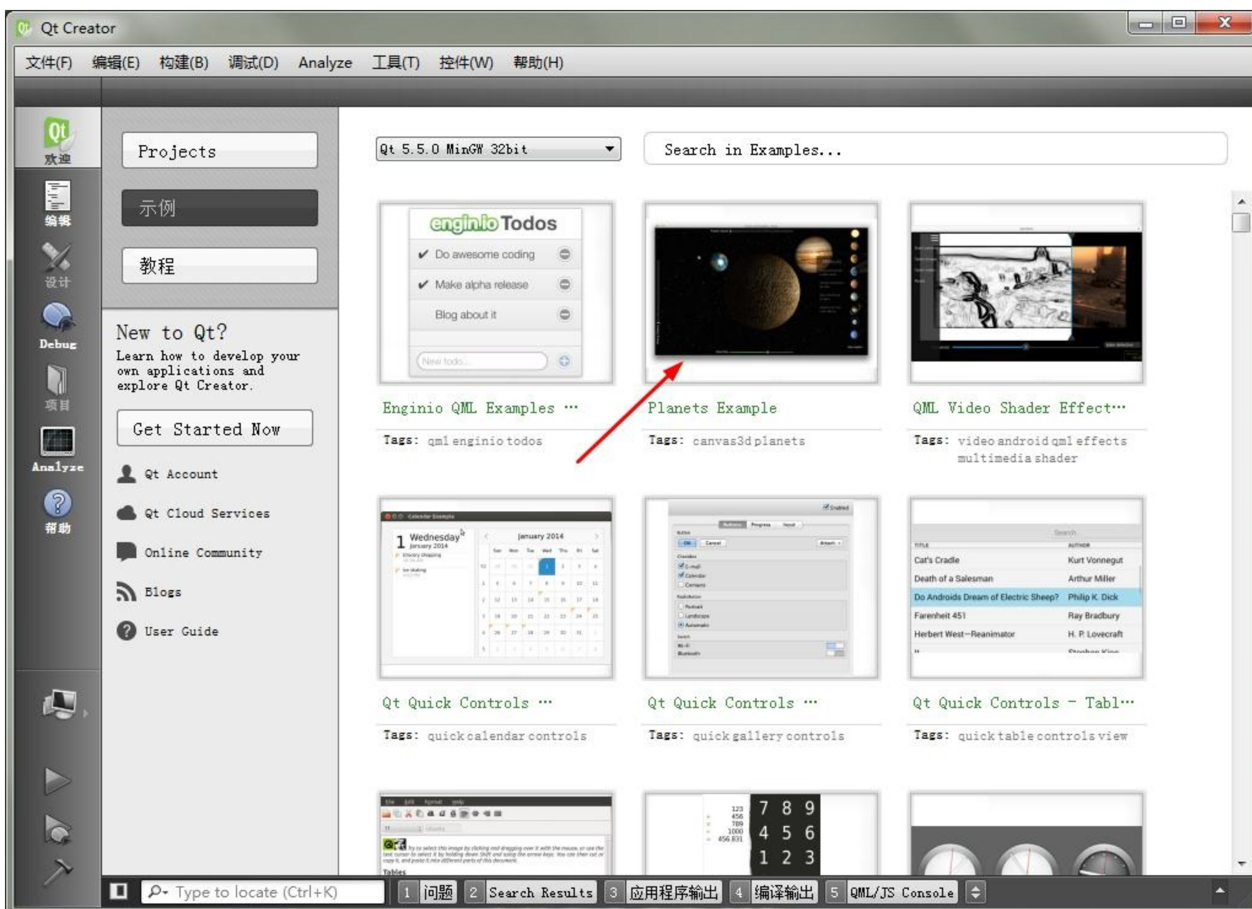
我们点击一个视频，发现它是放在YouTube上的，这样国内的同学也许无法直接访问，不过还是可以通过一些方式来登陆YouTube的，这个有兴趣的同学可以百度一下。视频教程的播放效果如下图所示。



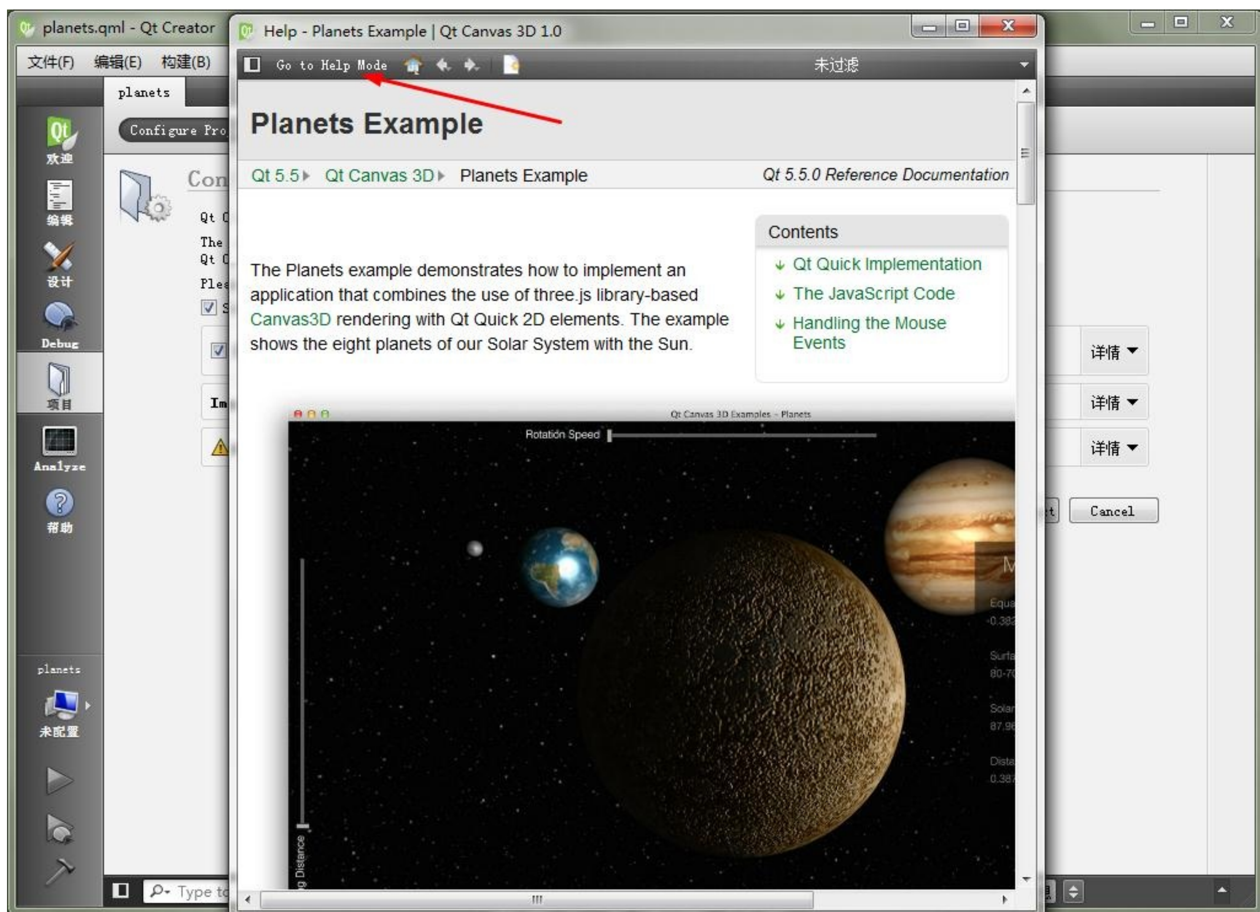
当然，视频教程是英文的，别说自己英文不好，编程里面除了数字符号，都是英文了，所以尝试接触英文教程是学好编程必须做的。

五、运行一个示例程序

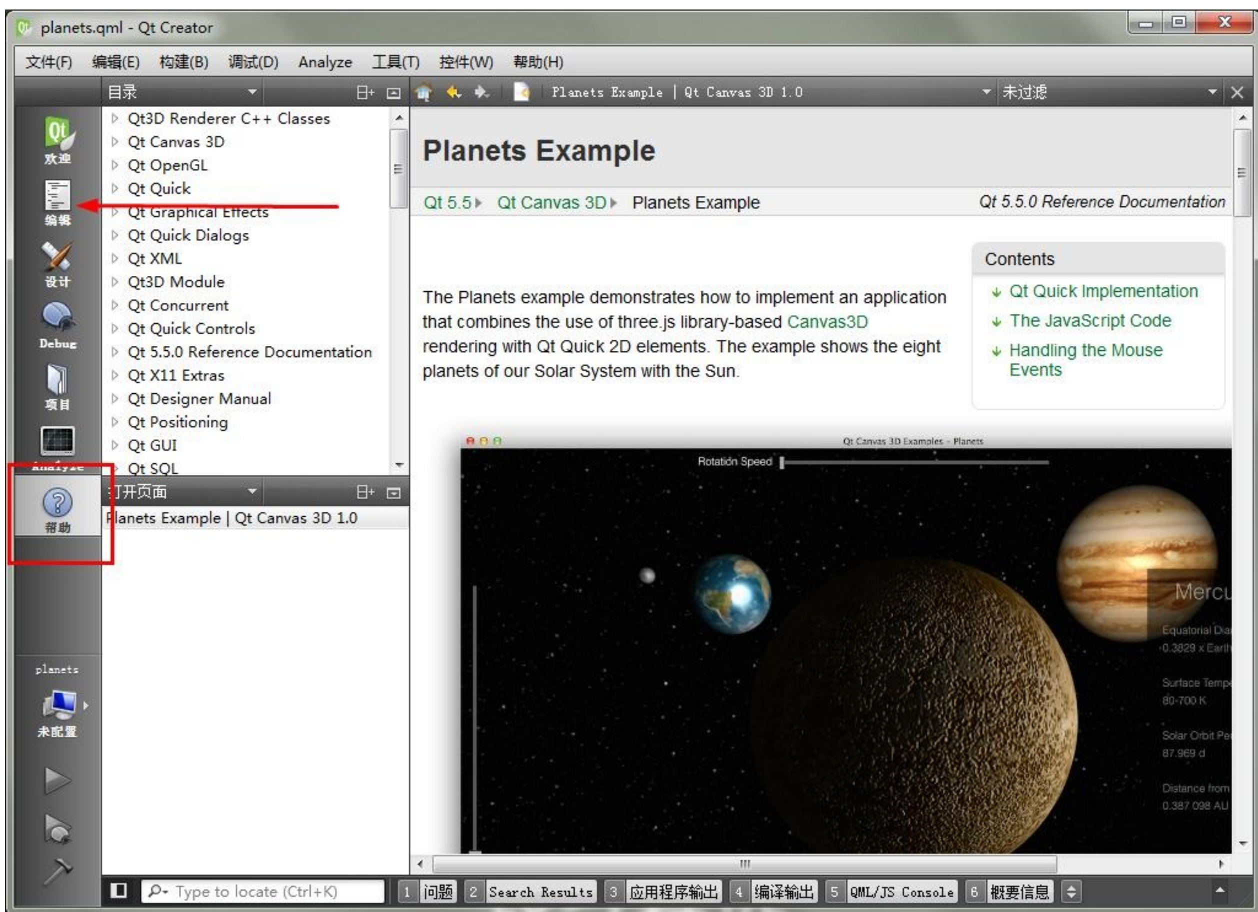
下面点击“示例”按钮，我们来运行一个Qt自带的示例程序。比如这里选择第二个Planets Example示例，它是Qt 5.5中新增的canvas3d演示程序，如下图所示。



这时会自动弹出该示例的帮助文档，如下图所示。



如果想在Qt Creator的帮助模式里查看该文档，可以直接点击上面的“Go to HelpMode”，这样该文档就会在帮助模式中显示。如下图所示。



现在在左下角可以看到planets项目还未配置，可以在项目模式中进行配置，如下图所示。



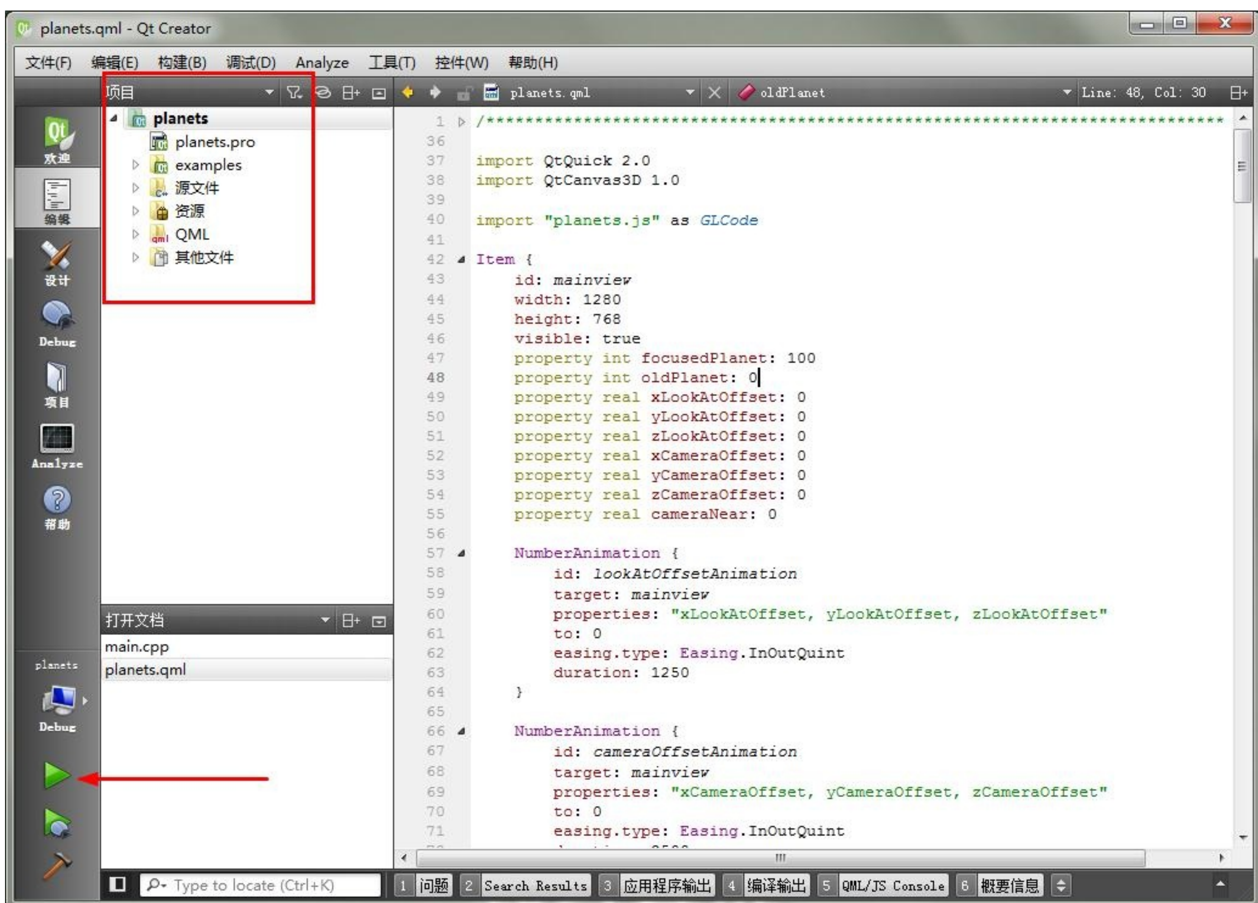
点击项目模式，可以看到默认选择了Desktop Qt 5.5.0MinGw套件，这表明是基于Qt5.5.0使用MinGw编译的桌面程序，也就是说安装好Qt 5.5，就自动为我们配置好了开发环境，可以直接使用。我们单击下面的ConfigureProject按钮来配置该项目。如下图所示。



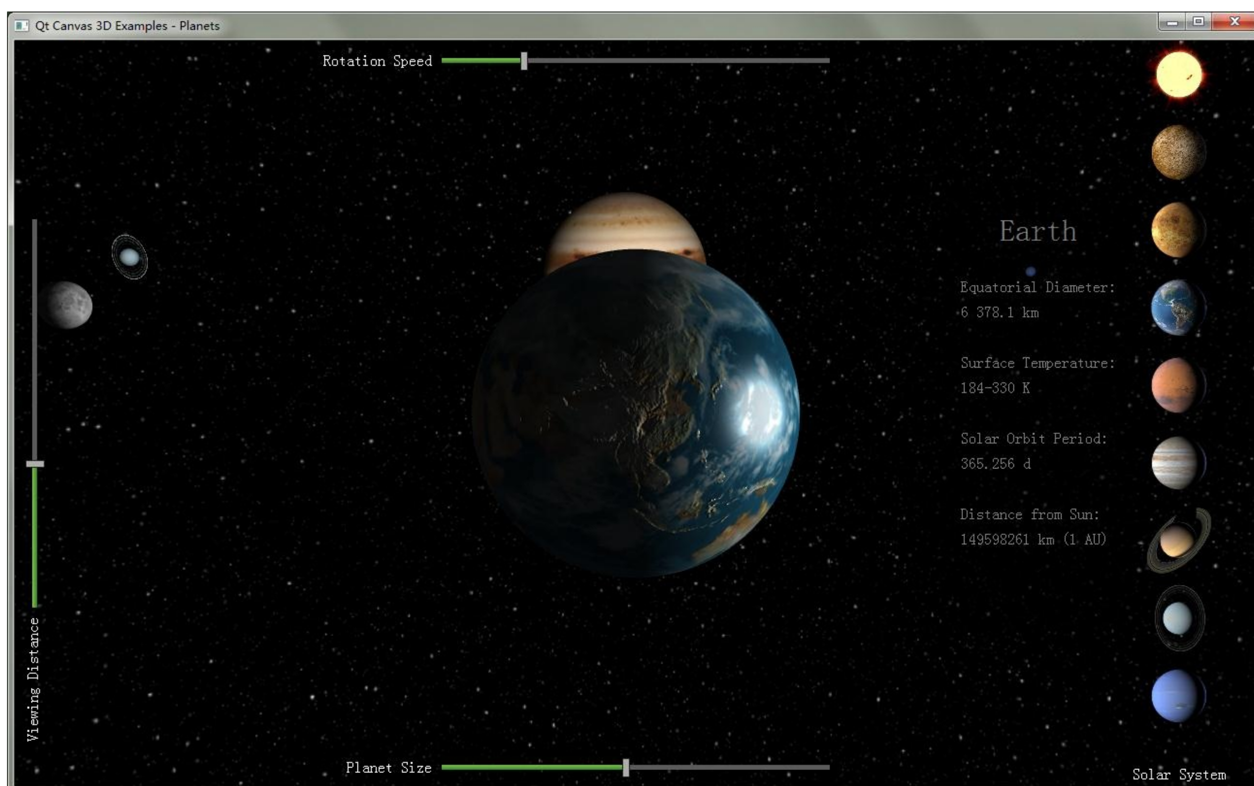
这时可以看到右下角读取项目的进度，如下图所示。



程序打开后，可以在左上角看到整个项目的源码列表。现在可以点击左下角的绿色三角运行按钮来编译运行该程序。如下图所示。



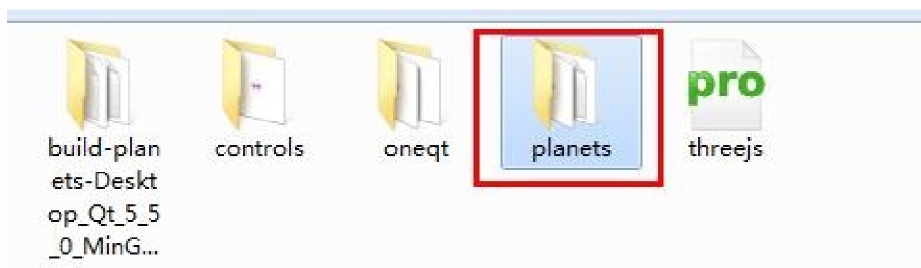
程序运行效果如下图所示。大家可以看到Qt 5程序的强大功能。



因为自带的示例程序是以后学习的参考，所以，如果想在程序上进行改动，最好在源码的备份上进行修改。要定位到该程序的源码，可以在列表文件上右击，然后点击“在Explorer中显示”，如下图所示。



现在看到程序的源码目录了，如下图所示，可以将其复制到其他位置，然后重新打开该项目，以后就可以按照自己的意愿修改该项目了，如果把项目修改的面目全非了，又想恢复到原始状态，只需要再次拷贝一下源码即可。



小结

从整体下载、安装并运行示例程序的过程看来，Qt 5.5版本与以前的版本变化并不大。在这一篇中我们下载并安装好了Qt SDK，然后通过一个示例程序测试了安装的Qt SDK可以使用，这样就完成了我们开发环境的搭建，从下一篇开始，我们将进入QtQuick的世界。

第52篇 Qt Quick简介

导语

在上一篇我们已经安装好了Qt 5.5，现在正式开始学习Qt5中全新的Qt Quick编程。Qt Quick对于大部分人来说是一个全新的概念，对这样一个全新的东西要怎样开始学习呢？在没有专业书籍（当然，《Qt 5编程入门》现在已经出版了，在这篇文章中我们假设读者没有Qt Quick编程经验），没有网络教程，需要自己开荒的情况下，我们的线索就是Qt帮助文档！

在这一篇中，我们会从Qt帮助入手，开始进入Qt Quick的世界，先讲解一些基本的概念，然后跟大家一起看一下最简单的Qt Quick项目是什么样的。让初学者先有一个大体的印象。

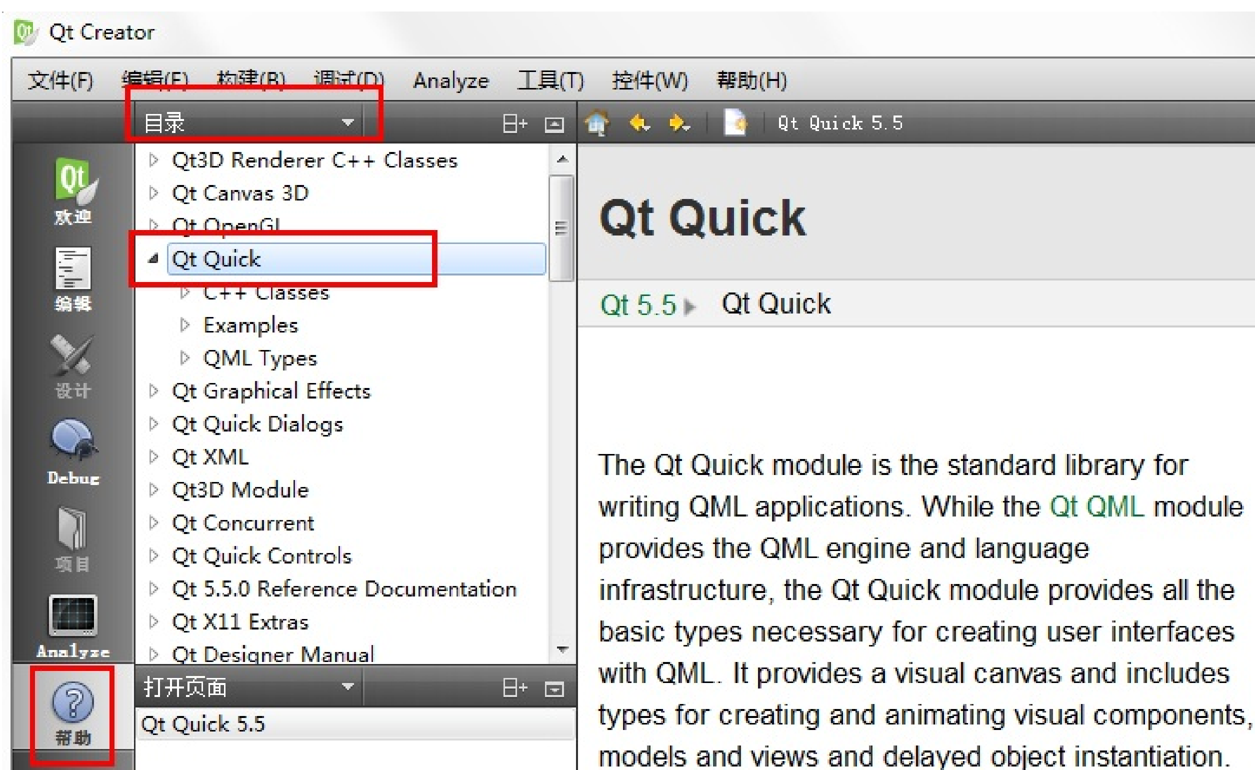
环境：Windows 7 + Qt 5.5.0+ Qt Creator 3.4.2

目录

正文

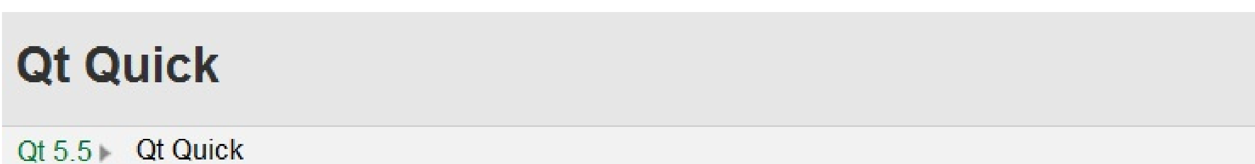
一、Qt Quick和QML简介

要学习Qt Quick编程，那么到底什么是Qt Quick呢？因为Qt Quick是Qt新引入的一个东东，所以要了解它，最好的方式就是查看Qt的官方文档。这里我们打开Qt Creator，然后点击进入帮助模式，在这里选择“目录”查看方式，这样可以显示出帮助文档中主要内容的目录。在这里我们选择“Qt Qucik”条码，如下图所示。



该文档讲述了什么是Qt Quick，以及Qt Quick所有相关的内容。可以看到，这里指出Qt Quick由Qt Quick模块提供，它是一个编写QML应用的标准库。Qt Quick模块提供了两种接口：使用QML语言创建应用的QML接口和使用C++语言扩展QML的C++接口。使用Qt Quick模块，设计人员和开发人员可以轻松地构建流畅的动态式QML用户界面，并且在需要的时候，可以将这些用户界面连接到任何C++后端。

说到这里，童靴们可能很想知道，那什么是QML呢？得益于Qt强大的帮助系统，所有相关的关键字都可以链接过去，我们可以点击最上面的“Qt QML”链接到其介绍页面，如下图所示。



The Qt Quick module is the standard library for writing QML applications. While the Qt QML module p
provides all the basic types necessary for creating user interfaces with QML. It provides a visual canv
receiving user input, creating data models and views and delayed object instantiation.

到该页面以后，可以看到，Qt QML模块为QML语言开发应用程序和库提供了一个框架。它定义并实现了语言及其引擎架构，并且提供了一个接口，允许应用开发者以自定义类型和集成JavaScript与C++代码的方式来扩展QML语言。Qt QML模块提供了QML和C++两套接口。如下图所示。

Qt QML

Qt 5.5 ▸ Qt QML

The Qt QML module provides a framework for developing applications and libraries with the QML language. It defines and implements the language and engine infrastructure, and provides an API to enable application developers to extend the QML language with custom types and integrate QML code with JavaScript and C++. The Qt QML module provides both a QML API and a C++ API.

这里介绍了Qt QML，并且提到了QML语言，我们进一步深入，看看QML语言到底是什么！点击最上面的“QML language”关键字。这时跳转到了“QML Applications”页面，这里对QML进行了定义。

QML（Qt Meta-Object Language，Qt元对象语言）是一种用于描述应用程序用户界面的声明式编程语言。它使用一些可视组件以及这些组件之间的交互来描述用户界面。QML是一种高可读性的语言，可以使组件以动态方式进行交互，并且允许组件在用户界面中很容易地实现复用和自定义。QML允许开发者和设计者以类似的方式创建高性能的、具有流畅的动画效果的、极具视觉吸引力的应用程序。QML提供了一个具有高可读性的类似JSON的声明式语法，并提供了必要的JavaScript语句和动态属性绑定的支持。QML语言和引擎框架由Qt QML模块提供。

到这里，也许大家对QML、Qt Quick有了一定的了解，也许有的读者可能对这些概念更加模糊了。这里举个可能不是很恰当的比喻，Qt Quick之于QML，正如Qt 之于 C++，QML是Qt中开发的一个新的语言，而Qt Quick是这个语言的一个组件库，其中包含了很多用QML写的可以现成使用的组件。大家暂且这样理解，对于一些内容也不要过于执拗，等后面见得多了，用的多了，慢慢就领悟了！

对于一个新的、庞杂的语言（或者说技术）要怎么学习，正如很多同学在问如何学习Qt，内容实在太多了！？其实很简单，答案是一步一步学习！这里没有开玩笑的意思，千里之行始于足下，大家都懂的道理，只是不愿付出行动而已。那有没有捷径？答案是“有”，就是从最简单的内容开始，从最标准的教程入手，以最规范的程序为例，保证一开始自己就是纯正血统，这样才不至于写出的程序乱七八糟，错误百出，浪费大量时间也找不出bug所在。

学习Qt，最标准的教程就是帮助文档，最规范的程序就是示例程序，而且如何开始学习，Qt文档中都给了入口，比如要学习QML语言，这里可以点击“First Steps with QML”链接，而要在Qt Creator中开发Qt Quick程序，可以参考“Creating Qt Quick Projects in Qt Creator”链接，如下图所示，本篇后面的内容就是按照这些文档来的。

Qt 5.5 ▸ QML Applications

QML is a declarative language that allows user interfaces to be described in terms of their visual components and how they interact and relate with one another. It is a highly readable language that was designed to enable components to be interconnected in a dynamic manner, and it allows components to be easily reused and customized within a user interface. Using the `QtQuick` module, designers and developers can easily build fluid animated user interfaces in QML, and have the option of connecting these user interfaces to any back-end C++ libraries.

What is QML?

QML is a user interface specification and programming language. It allows developers and designers alike to create highly performant, fluidly animated and visually appealing applications. QML offers a highly readable, declarative, JSON-like syntax with support for imperative JavaScript expressions combined with dynamic property bindings.

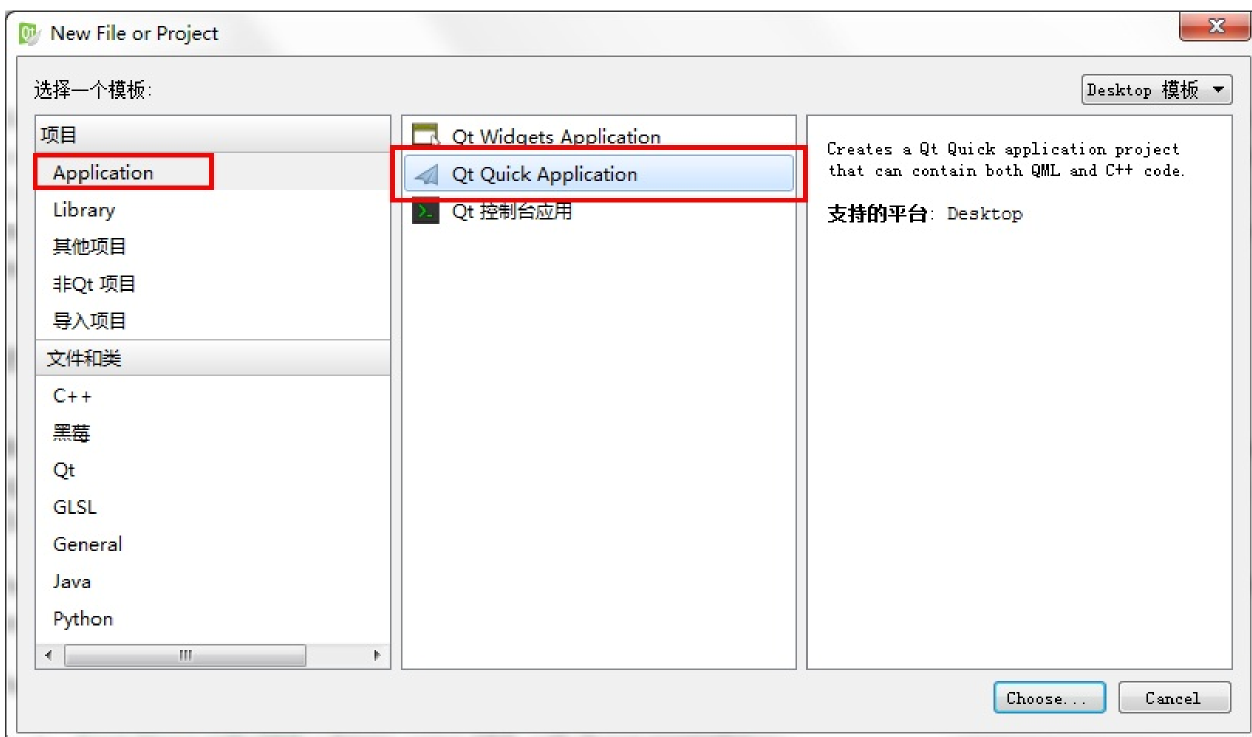
The QML language and engine infrastructure is provided by the Qt QML module. For in-depth information about the QML language, please see the Qt QML module documentation.

The following pages contain more information about QML:

- First Steps with QML - begin using QML with these examples
- Creating Qt Quick Projects in Qt Creator

二、创建一个Qt Quick应用

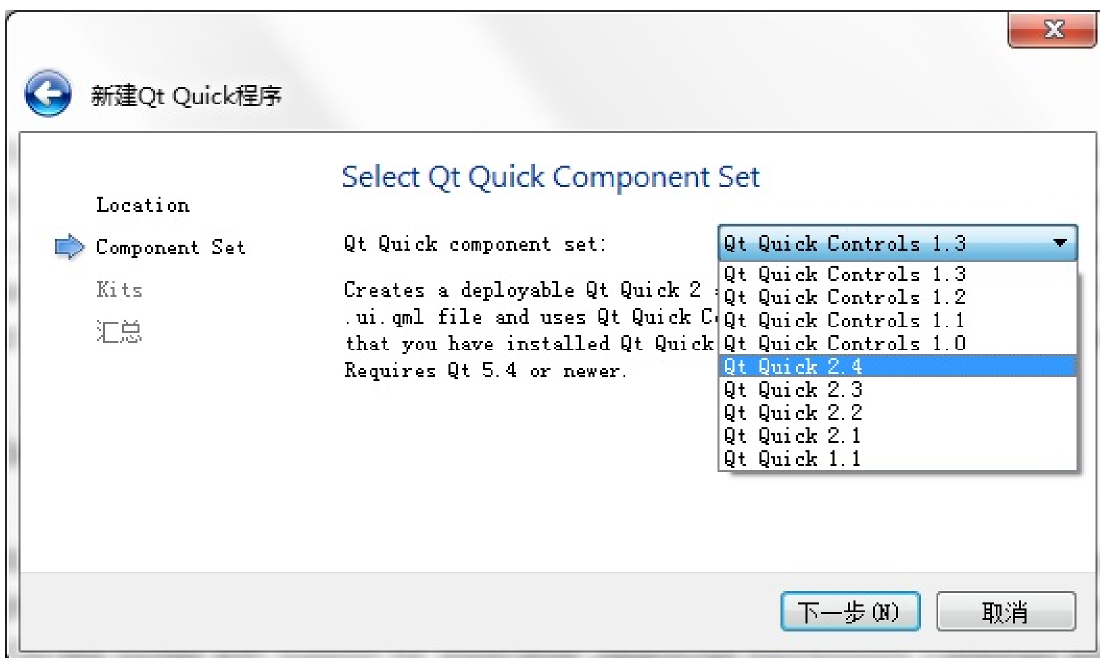
打开Qt Creator，点击“文件→新建文件或项目”菜单项，然后选择创建“Qt QuickApplication”，如下图所示。



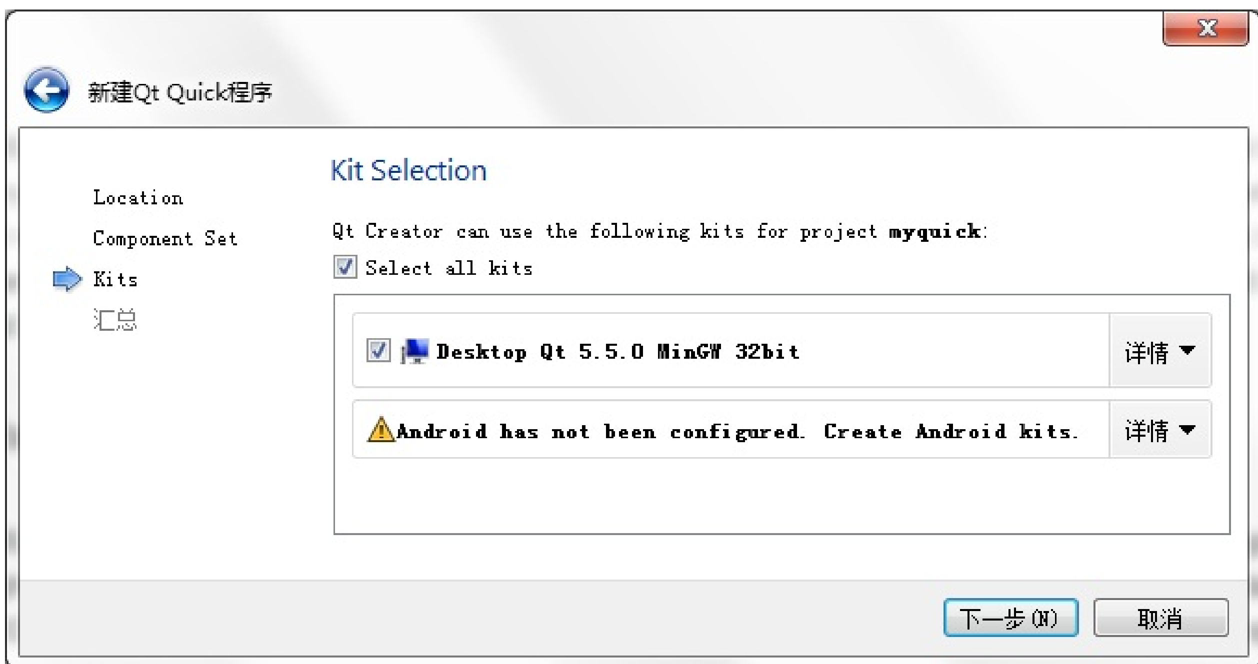
在项目介绍和位置界面，设置好项目名称和源码路径，注意路径中不要包含中文。如下图所示。



下面是选择Qt Quick组件集，默认是使用Qt Quick Controls 1.3组件集，Qt Quick Controls是Qt Quick通用控件模块，这个会在后面的内容里重点讲解，由于我们初学Quick，为了不迷茫于众多新名词、新概念的大海中，这里先不介绍这个。我们选择下面的“QtQuick 2.4”，它包含了基本的QML类型，2.4是现在的版本号。如下图所示。



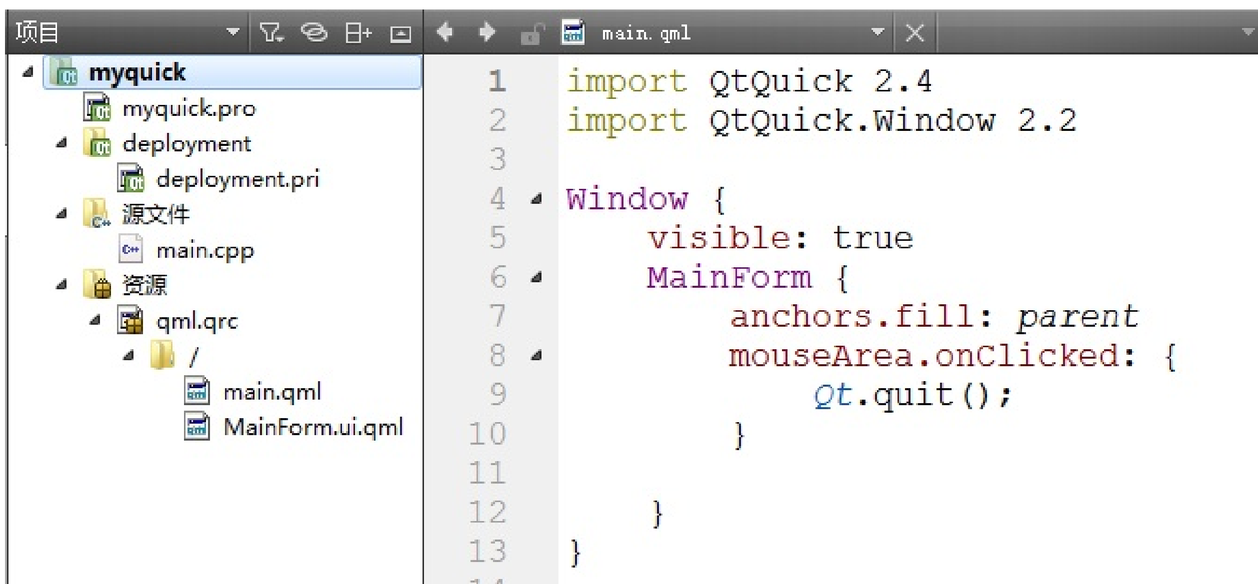
然后是构建套件选择，这里使用默认的Qt 5.5版本即可（如果大家的电脑上安装了多个Qt版本，可能这里会显示多个选择，因为我们现在讲的是Qt 5.5，所以只选择这个即可）。如下图所示。



最后是项目管理选项，因为我们是单独的项目，也不需要版本控制，所以直接点击“完成”即可。如下图所示。



项目创建好以后，会自动在编辑模式显示 `main.qml` 文件的内容，而在左边项目文件列表中分类显示了该项目中的所有文件。如下图所示。



我们先来运行一下这个程序，看下运行效果。按下 `Ctrl+R` 快捷键，或者点击左下角的绿色三角运行图标来编译运行程序，效果如下图所示。可以看到，这是一个最简单的Hello World程序，在一个空白窗口中显示了“Hello World”字符串，当在窗口中点击鼠标时，窗口会关闭，程序退出。



三、QML文件内容介绍

看到了运行结果，现在我们来简单看下这个 `main.qml` 文件的内容，后缀 `.qml` 表明这是个QML文件，其内容如下：

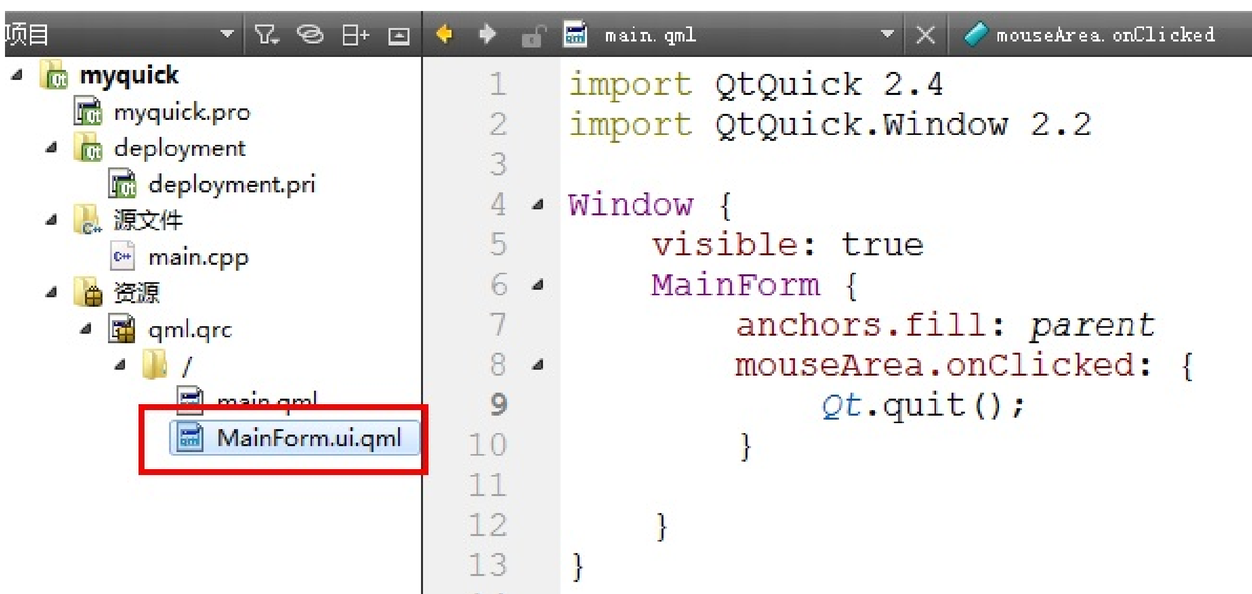
```
// 导入语句部分
import QtQuick 2.4
import QtQuick.Window 2.2
//对象声明部分
Window {
   visible: true
    MainForm {
       anchors.fill: parent
       mouseArea.onClicked: {
            Qt.quit();
       }
   }
}
```

正如这段代码所示的，一个QML文档定义了一个QML对象树，由两部分组成：一个 `import` 导入部分，一个对象声明部分。

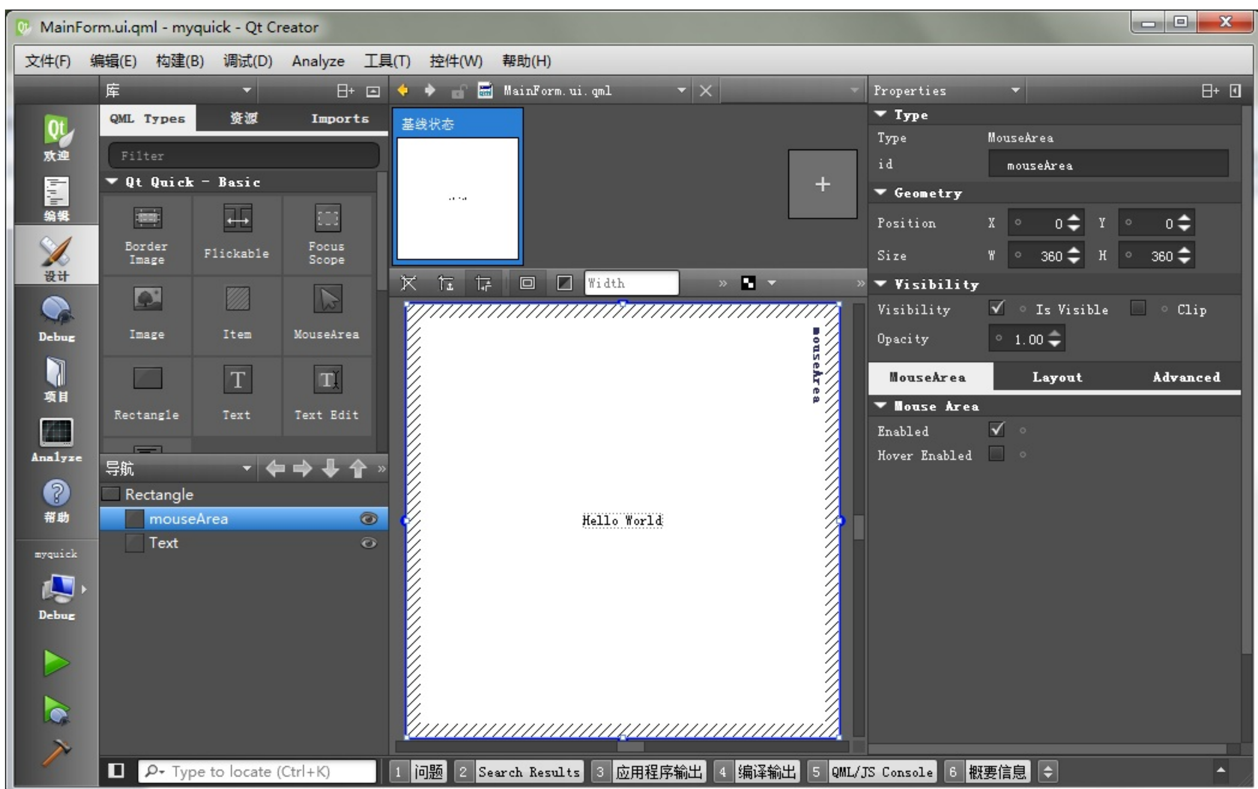
`import` 导入语句类似于C++中的#include，只有导入了相关模块，才能使用其中的类型和功能。这里导入了QtQuick模块，这个就是我们前面创建项目时选择的组件集，它包含了创建用户界面所需要的基本类型和功能；而 `QtQuick.Window` 模块中提供了Window类型，它可以为Qt Quick场景创建一个顶层窗口。

下面再来看对象声明部分，QML文档中的对象声明定义了要在可视场景中显示的内容。这里创建了两个对象： `Window` 对象和其子对象 `MainForm` 。对象（object）由它们的类型（type）指定，以大写字母开头，后面跟随一对大括号，在括号中包含了对象的特性定义，比如这个对象的属性值或者其子对象。最外层的对象叫根对象，比如这里的 `Window` ，在根对象里面定义的对象，叫做根对象的子对象，比如这里 `MainForm` 就是 `Window` 的子对象。在 `Window` 中的 `visible` 是 `Window` 的属性，用来设置窗口是否显示，可以在帮助文档中查看一个类型的所有属性及用法。

再来看 `MainForm` ，它不是 `QtQuick` 模块中的类型，而是自定义的一个用户界面表单（Qt Quick UI Forms），这是Qt 5.4以后提出的一个概念，类似于Qt C++编程中的UI文件，大家看下左边文件列表里面的 `MainForm.ui.qml` ，就是这个文件，如下图所示。

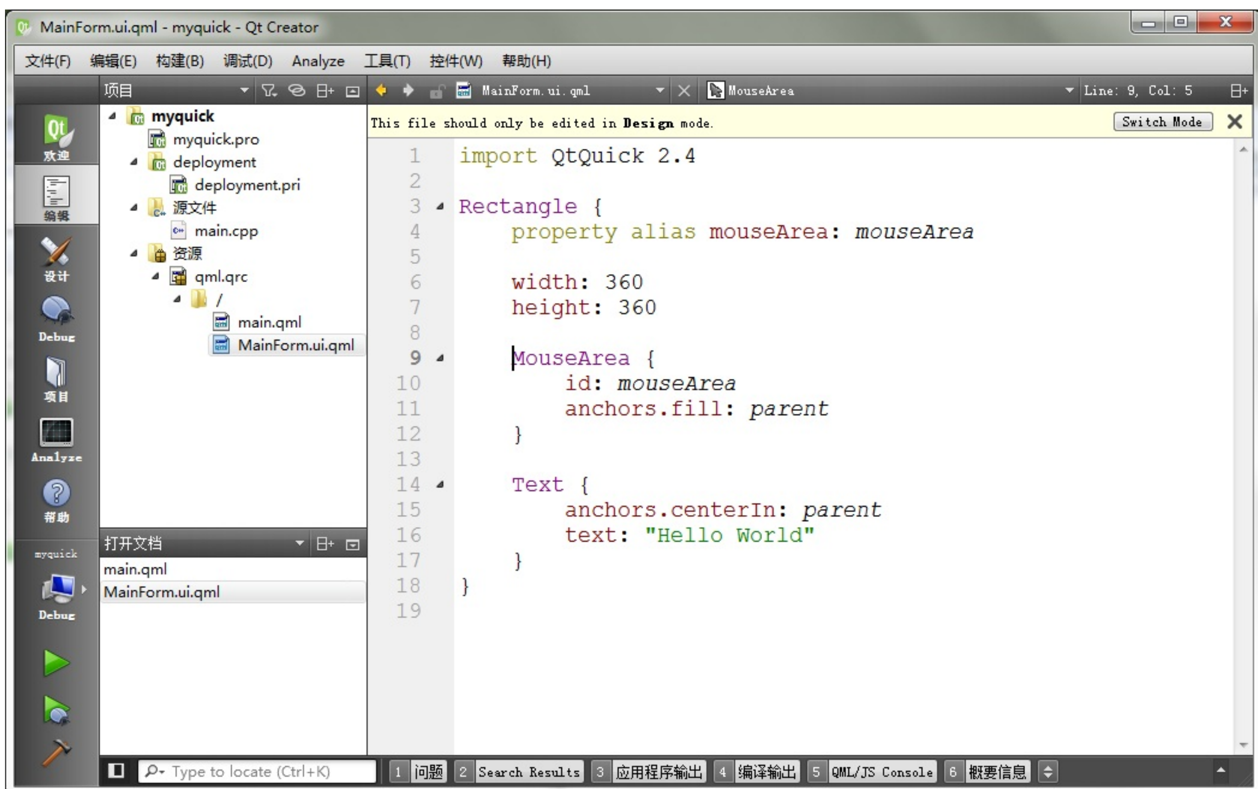


我们双击这个文件，这时会自动跳转到设计模式，也就是所谓的Qt Quick设计器。如下图所示。



在设计器中可以实现所见即所得的效果，可以弱化编码操作，专心进行设计，而后在文件中通过编码实现逻辑操作，这样也可以很大程度地把设计和逻辑相分离。Qt一直致力于改进Qt Quick设计器，比如现在这个版本就极力推荐使用设计器。但是，很多特性在设计器中还是没有办法实现的，依然需要使用代码来完成，并且，现在实际工作中还不是设计人员与编码人员的完全分离，一般我们还是程序员一个人来全部搞定。所以，实际中设计器用的并不多。

可以看到设计器中设计的界面就是程序运行显示的界面，现在我们直接按下`Ctrl+2`快捷键，或者点击“编辑”模式，可以看到设计器中设计的界面对应的代码，如下图所示。



在最上面有一行提示“This file should only be edited in Design mode”，就是该文件只应该在设计模式中被修改，其实这里是可以直接修改代码的。这里 `MainForm.ui.qml` 文件的内容如下：

```
// 导入语句
import QtQuick 2.4

// 根对象是一个矩形
Rectangle {
     //声明一个新的mouseArea属性，作为子对象MouseArea的id属性的属性别名
      property alias mouseArea: mouseArea
      // 定义矩形的宽和高
       width: 360
       height: 360

       // MouseArea是一个不可见项目，为它覆盖的可见项目提供鼠标处理
       MouseArea {
                //id属性作为该对象的标识，在其他地方可以通过id来引用该对象
                id: mouseArea
                // 这个属性起到布局的作用，这里就是填充整个父项目
                 anchors.fill: parent
          }

         //Text是文本项目，用来显示文本内容
         Text {
                // 使文本项目处于父项目的中心位置
                anchors.centerIn: parent
                // 文本内容是“Hello World”
                text: "Hello World"
          }
}
```

这段代码定义了一个矩形项目（在代码中，从语法角度来说 `Rectangle` 是对象，但是，从界面角度来说，叫对象不合适，我们这里叫做项目，或者会叫做组件，由于名词术语的冲突，统一把界面上可视的组件叫做项目，这个后面深入讲解），然后在矩形上面遮了一个鼠标区

域`MouseArea`，这样整个矩形都可以接受鼠标操作了，最后在矩形的中间放了一个文本项目，显示了helloworld字符串。

这里需要说明一下，在开始用“property alias”声明了一个属性`mouseArea`，在根对象中声明的属性可以在其他QML文档中使用，而且“property alias mouseArea: mouseArea”后面一个`mouseArea`表明了它是子对象`MouseArea`的`id`属性的属性别名，也就是说在其他QML文档中操作`mouseArea`属性，就相当于操作这里的`MouseArea`对象。现在再回过头来查看`main.qml`文件的内容：

```
Window {
    visible: true
    MainForm {
        anchors.fill: parent
        mouseArea.onClicked: {
            Qt.quit();
        }
    }
}
```

这里的`MainForm`就是`MainForm.ui.qml`的一个实例，而在它里面调用了`mouseArea`属性的`onClicked`事件处理器，这就相当于调用了`MainForm.ui.qml`中的`MouseArea`对象的`onClicked`事件处理器，其中`Qt.quit()`表明，当在整个矩形中点击鼠标时要执行的命令就是退出程序。

到这里，主要的两个`qml`文件都讲解完了，很多读者应该会感觉很乱，新的名词、新的内容还是太多，不知从何处下手。没有关系，这一篇中我们只是对创建的整个项目内容进行介绍，让大家有个初步的了解，在下一篇中我们会从最简单的内容开始讲起，然后一点点进行扩展！

四、其他文件内容介绍

讲完了主要的QML文件，下面我们再来看一下项目中的其他文件。整个项目中，所有的QML文件是以资源的形式放在`.qrc`文件中的，大家可以猜测QML文件跟C++源文件是不同的，它们并不需要编译，而更类似于素材。下面来看一下`main.cpp`文件的内容：

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    return app.exec();
}
```

这个文件内容很简单，主要就是在主函数中定义了一个`QQmlApplicationEngine`对象，并用其加载了`main.qml`文件。`QQmlApplicationEngine`类结合了`QQmlEngine`和`QQmlComponent`两个类的功能，提供了一个便捷的方式来加载一个QML文件，但这个QML文件的所有可视内容必

须放在 `Window` 对象中才能最终显示出来。对于相关的内容我们这里不再深入讲解，大家现在只需要知道，以后在 `main.cpp` 文件中通过这种方式来调用QML文件即可。

然后是 `myquick.pro` 项目文件，其中就是简单指明了程序的模板、使用的模块、源文件、资源文件等，这个与C++项目类似。还有一个 `deployment.pri` 文件，它是项目文件的补充内容，其中指出了编译到不同平台的设置信息。这些内容现在我们也不需要深入了解，这里就不再展开来说了。

小结

这一篇中我们从学习新东西（或者说新的语言、新的技术）的最自然的角度开始，引领初学者进入Qt Quick编程的大门，帮助初学者掌握学习的技巧，学习Qt的方式。这也是网络教程与书籍的不同之处，在《Qt 5编程入门》中我们是以另外一种方式讲述的，读者的喜好不同，可以选择自己感兴趣的方式学习，或者两者都学习一下！

该篇中我们全面讲解了Qt Quick项目的所有内容，虽然没有深入，但是让初学者有了一个感性的认识，这个认识会在后面的学习过程中逐渐强化，从陌生到熟悉，最后达到灵活运用。

第53篇 Qt Quick项目详解

导语

前面我们一起创建了一个Qt Quick项目，并对里面的文件进行了简单的讲解，虽然这只是一个HelloWorld程序，但对于没有Qt Quick编程经验的同学来说，这个项目还是有点复杂。在这一篇中，我们将从最简单的QML文件讲起，然后逐渐丰富项目内容，帮助大家由浅及深的进行学习，进一步了解Qt Quick项目的构成。

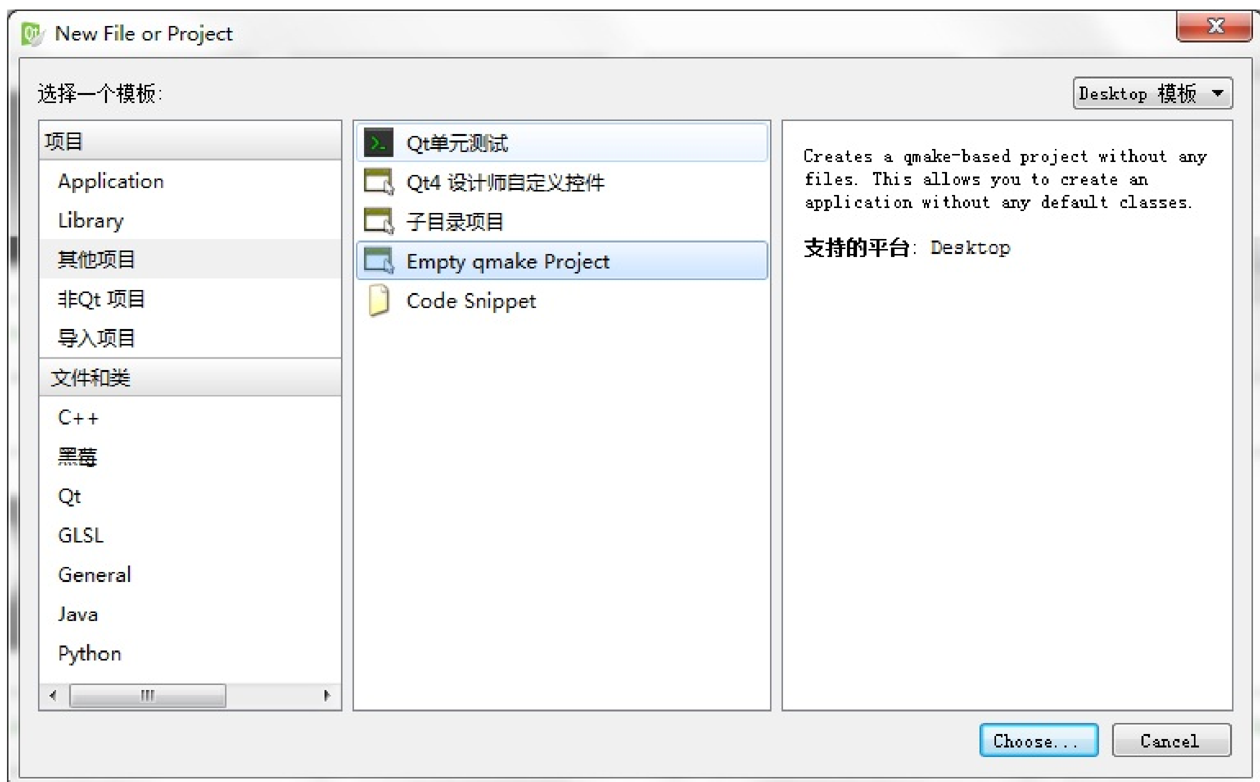
环境：Windows 7 + Qt 5.5.0+ Qt Creator 3.4.2

目录

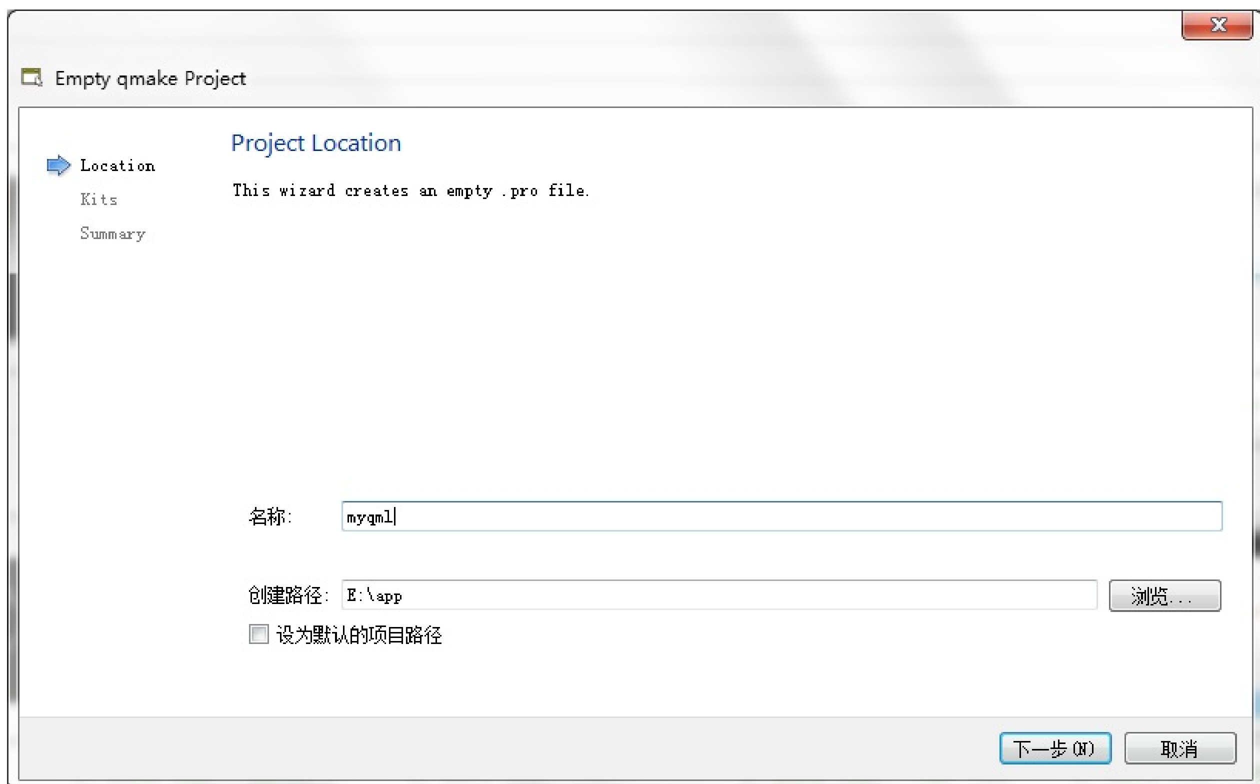
正文

一、创建空项目

1、首先打开Qt Creator，然后选择“新建文件或项目”菜单项，在选择模板页面选择“其他项目”分类中的“Empty qmake Project”，我们先来创建一个空项目，后面逐步往里面添加文件。点击“choose”按钮确定选择，如下图所示。



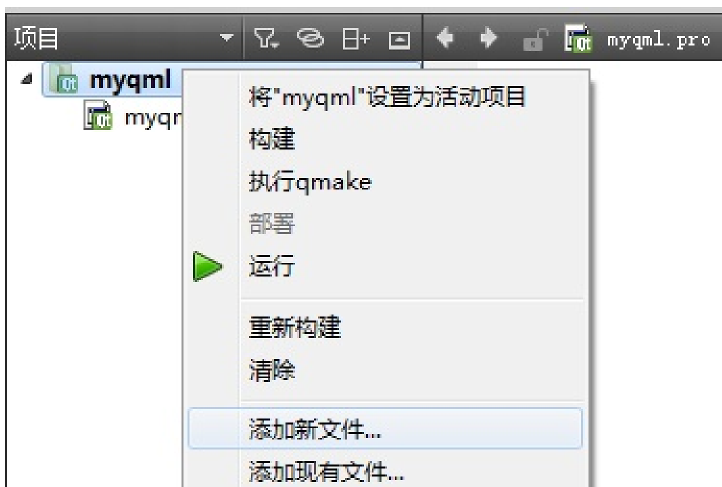
2、在项目位置页面，将项目名称修改为 `myqml` ，然后选择好创建路径。如下图所示。



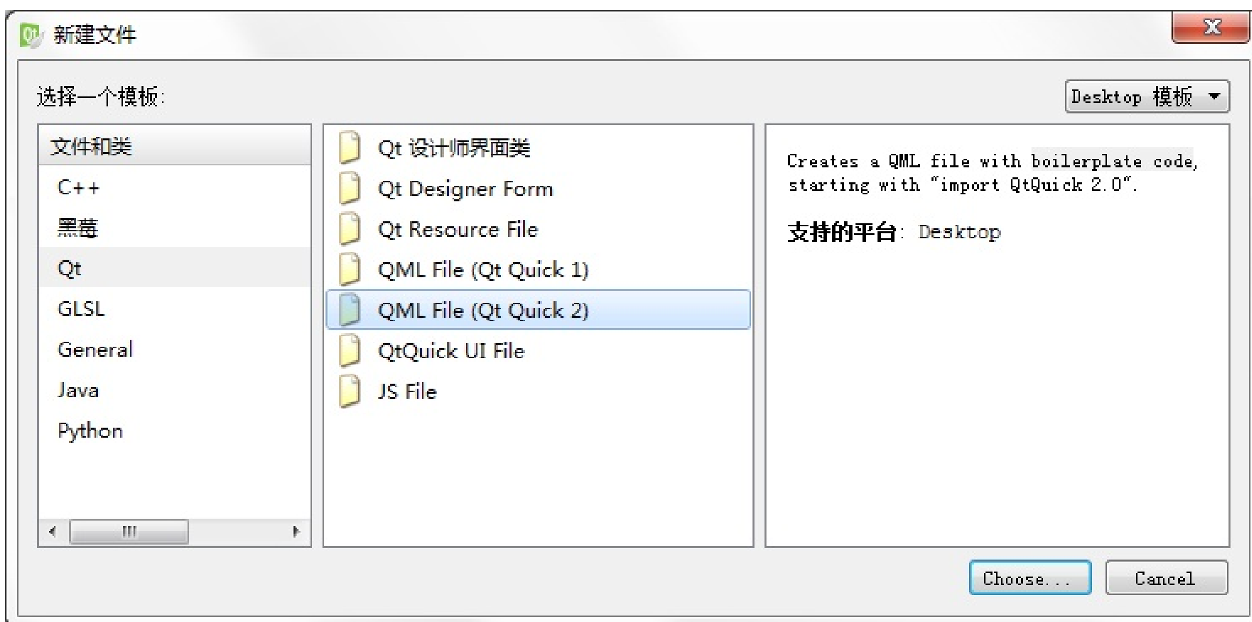
3、后面的步骤直接使用默认设置即可。

二、添加QML文件

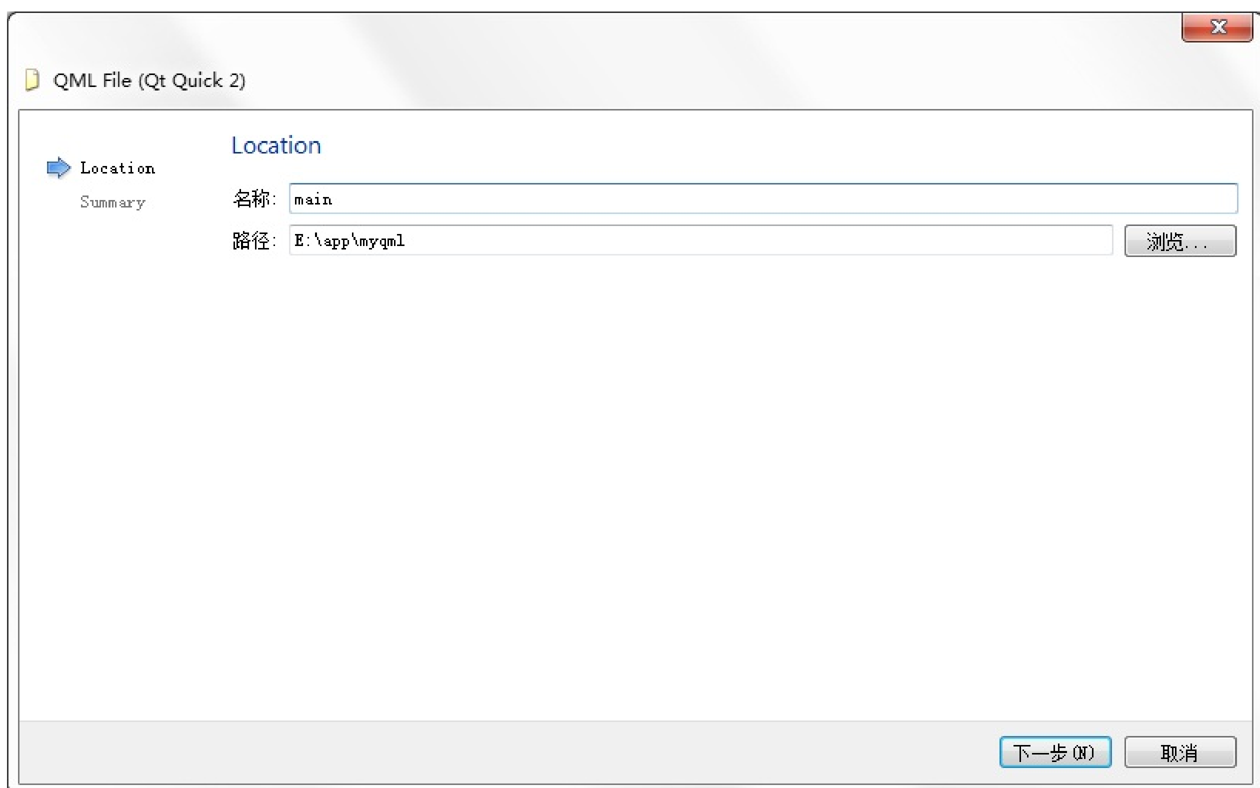
1、创建好的空项目只有一个 `.pro` 项目文件，现在里面的内容也是空的。我们先不去管它，下面向该项目中添加文件。在 `myqml` 文件夹上右击，在弹出的菜单中选择“添加新文件”，如下图所示。



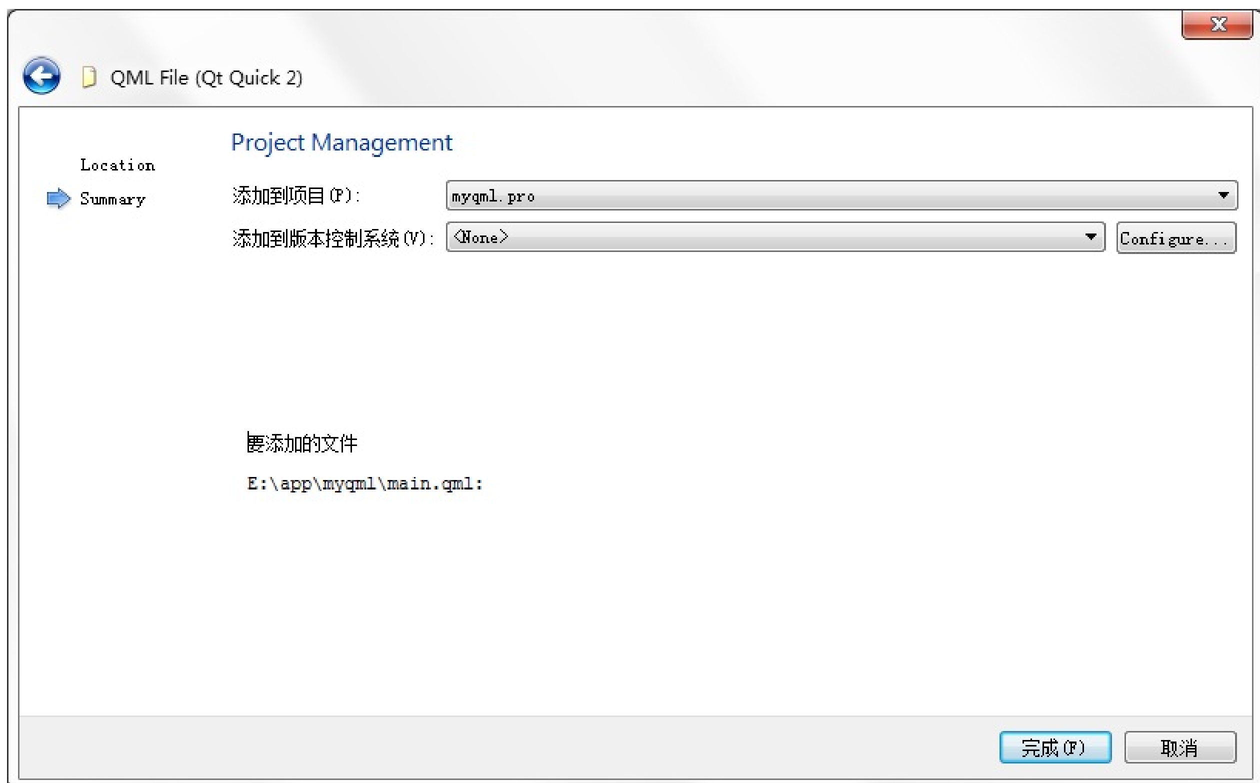
2、在弹出的新建文件对话框中选择Qt分类中的“QML File（Qt Quick 2）”一项。如下图所示。（说明一下：这里要添加QML文件，模板可以选择Qt Quick 1、Qt Quick 2和QtQuickUI File三种，Qt Quick 1会导入QtQuick 1.1模块，里面的内容都是Qt 4时代的；而QtQuick 2导入的是QtQuick 2.0模块，是Qt 5中的新版本；QtQuick UI文件生成后会默认使用设计器，因为我们这一节直接讲代码，不涉及设计器的内容，所以不选择这个模板。其实，使用哪个模板都一样，因为到文件里面可以直接修改导入模块语句，初学者在这里不要纠结。）



3、下面设置文件的名称为 `main` ，路径保持默认即可，现在的路径就是项目源码路径。然后点击“下一步”按钮，如下图所示。

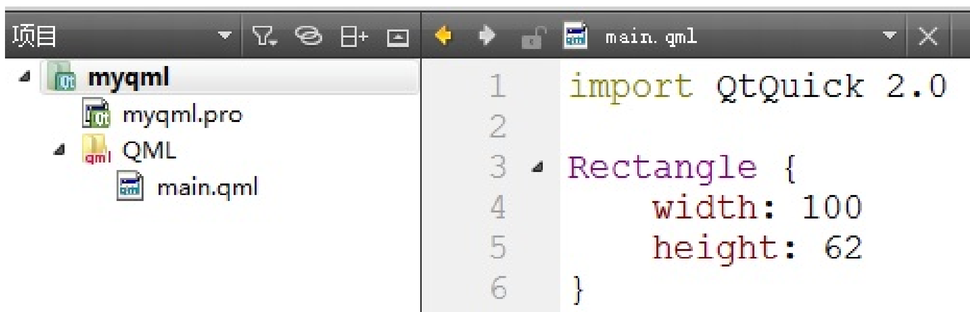


4、最后是项目管理页面，可以看到，新创建的文件默认添加到了 `myqml.pro` 项目中，直接点击"完成"按钮完成文件的添加建，如下图所示。



三、运行程序

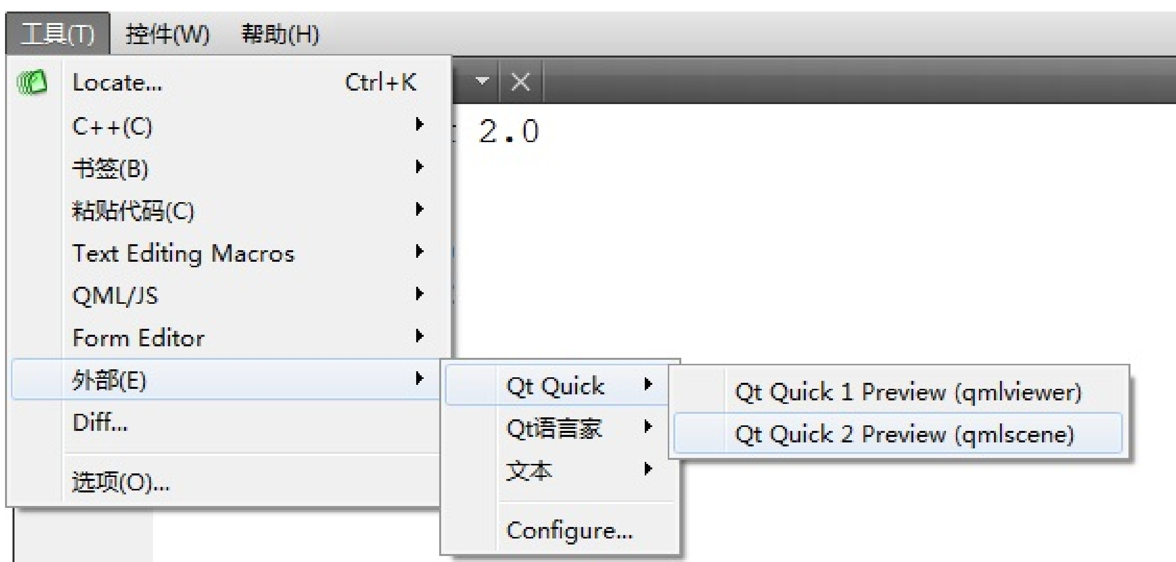
1、可以看到添加的文件就是 `main.qml` ，后缀 `.qml` 表明该文件是一个QML文件，其内容如下。



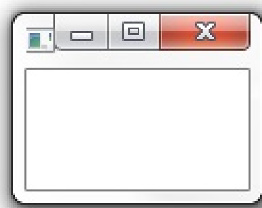

```
import QtQuick 2.0
Rectangle {
    width: 100
    height: 62
}
```

这是一个最简单的QML文件，它会显示一个宽100像素高62像素的矩形。这里的 `import` 语句上一节已经提到过了，它会导入相应的模块，比如这里导入了QtQuick 2.0模块。下面的 `Rectangle` 是矩形对象，用来定义一个矩形项目（项目类似于C++中的窗口或者部件），里面的 `width` 和 `height` 是 `Rectangle` 的属性，用来设置矩形相关参数。

2、QML文件与C++文件是不同的，它不需要进行编译，可以直接运行。在Qt中提供了两个运行QML文件的工具qmlviewer和qmlscene，前者是Qt 4时代的产物，主要用来显示导入了QtQuick 1.1模块的QML文件，而qmlscene用来显示导入了QtQuick 2.0以后版本的QML文件。选择“工具→外部→QtQuick→Qt Quick 2Preview”菜单项即可在qmlscene中显示现在打开的QML文档的内容。如下图所示。



运行效果如下图所示。



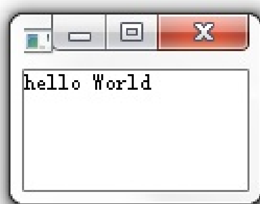
四、扩展QML程序

1、添加文本显示

可以看到现在的程序就是一个空白的窗口，什么都没有。下面我们来扩展该程序，将 `main.qml` 文件的内容更改如下：

```
import QtQuick 2.0
Rectangle {
    width: 100
    height: 62
    Text {
       text: "hello World"
    }
}
```

这里添加了一个 `Text` 对象，它用来显示一块文本，其 `text` 属性用来指定要显示的文本内容。下面先按下 `Ctrl+S` 快捷键保存该文件，然后在qmlscene中查看显示效果，如下图所示。



2、使用锚布局

我们看到现在可以显示hello world文本了，但是文本默认显示在了左上角，我们希望它可以在窗口中间显示，下面继续添加代码：

```
Text {
       text: "hello World"
       anchors.centerIn: parent
    }
```

这里使用了 `anchor` 锚的概念进行了布局，在QML中每一个项目都有一组无形的锚，分别在上、下、左、右、中心等处，它们可以定义项目自身和其他项目的相对位置。比如这里的 `centerIn` 就是指 `Text` 项目在 `parent` 项目的中心，这里的 `parent` 就是指 `Text` 的父项目 `Rectangle` 。在 `anchor` 后面一般指定其他项目的 `id` ，如果是父子项目，也可以像现在这样指定 `parent` 。

下面先保存代码（以后每次修改都需要保存后才能显示，后面不再提醒），然后在qmlscene中运行，效果如下图所示。



3、添加鼠标互动。

前面的代码已经完成了一个简单的Hello World界面，下面我们添加代码实现点击界面退出程序的效果：

```
import QtQuick 2.0
Rectangle {
    width: 100
    height: 62
    Text {
       text: "hello World"
       anchors.centerIn: parent
    }
    MouseArea {
       anchors.fill: parent
       onClicked: {
           Qt.quit()
       }
    }
}
```

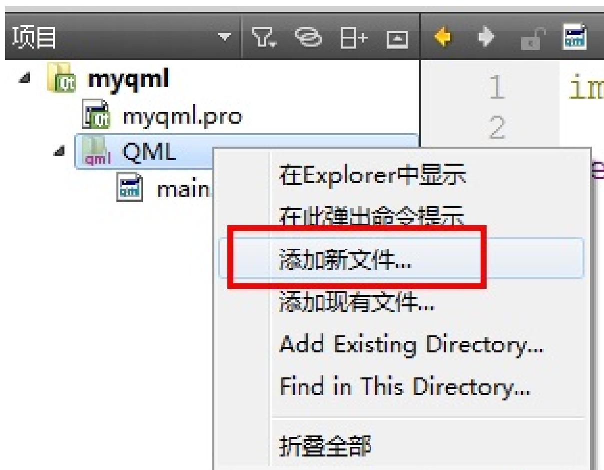
我们在 `Rectangle` 中又添加了一个 `MouseArea` 子对象，这个从字面上翻译就是鼠标区域，它是一个不可见的项目，就是说我们在窗口上并不能看到它的存在，通过该对象可以实现鼠标互动。 `anchors.fill` 是进行填充，这里就是将鼠标区域覆盖整个 `Rectangle` 窗口。 `onClicked` 其实就是Qt C++中的信号处理函数，这里一般叫做信号处理器，其语法是 `on<Signal>` ，所以这里就是 `Clicked` 单击信号的处理，当在窗口上单击鼠标后会执行 `Qt.quit()` 函数，这是个全局函数，执行结果就是使程序退出。

大家可以在qmlscene中运行程序，然后在窗口上点击鼠标查看运行效果。

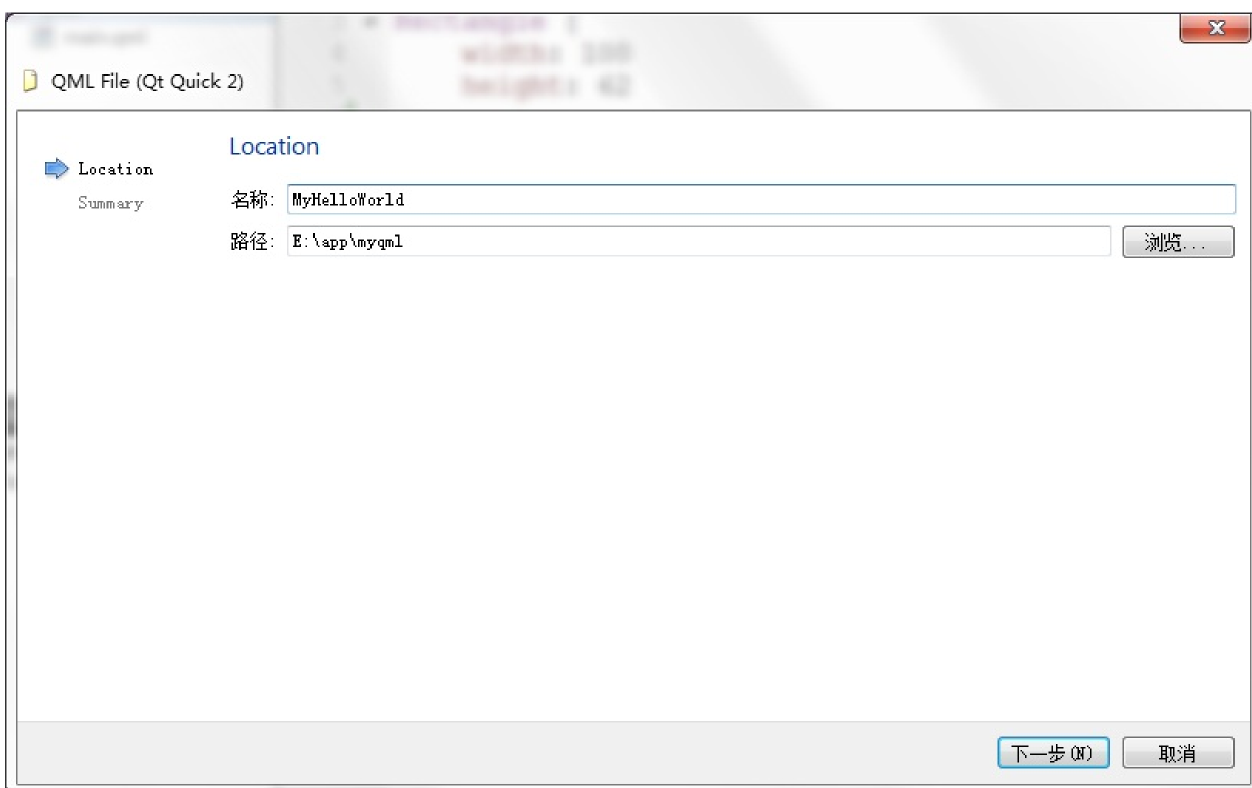
4、使用组件

前面创建的是 `main.qml` 文件，本意是让其作为主要文件，成为程序的入口。但是如果按照现在的做法，程序越写越复杂， `main.qml` 的内容会越来越多，越来越乱。为了避免在 `main.qml` 中添加过多的代码，我们一般将具体的实现代码放到单独的文件中。

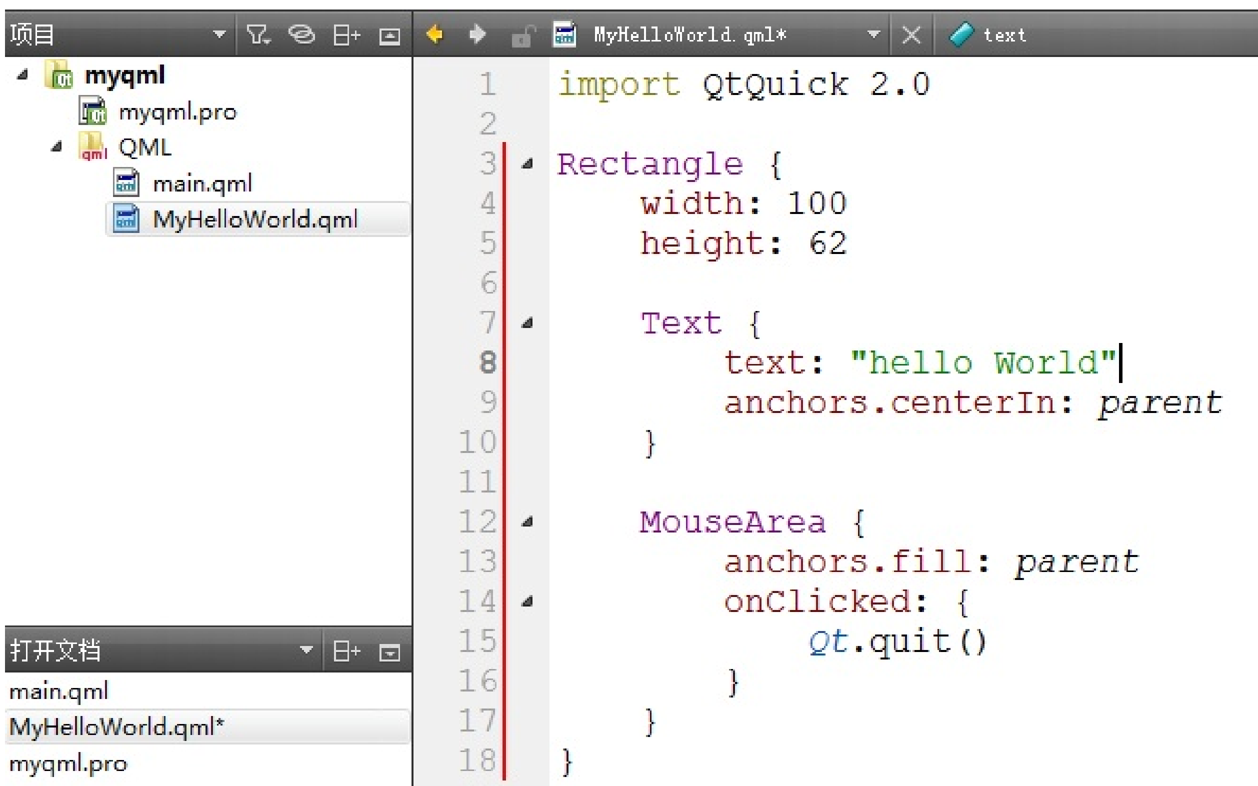
下面首先在QML目录中添加新文件，如下图所示。



模板依然选择Qt分类中的QML File（QtQuick 2），文件名称设置为 `MyHelloWorld` ，请注意首字母要大写，如下图所示。



添加完成后，将 `main.qml` 中的内容全部复制粘贴过来，如下图所示。



下面我们修改 `main.qml` 文件的内容如下：

```
import QtQuick 2.0
Item {
    MyHelloWorld {
        anchors.fill: parent
    }
}
```

像这样在一个单独文档中的QML代码段，就定义了一个对象类型，它也叫做组件，这个文件的名称必须以大写字母开头。比如这里创建了一个 `MyHelloWorld` 类型，在其他QML文件中可以直接使用同一目录下自定义的对象类型，所以，我们在 `main.qml` 中直接创建了 `MyHelloWorld` 对象。这里根对象使用了 `Item` ，在Qt Quick中，所有可视项目都继承自 `Item` ，因为在 `main.qml` 中我们只需要创建一个窗口，不需要进行内容设置，所以一般使用 `Item` 即可，当然，如果想使用 `Rectangle` 等也是可以的。

使用组件除了可以简化代码外，更重要的是它可以被重复使用，如果在一个代码中需要多次使用相同的部件或者功能，将它们作为组件就没必要多次编写相同的代码了。大家可以保存文件后在qmlscene中运行程序，查看效果。

5、 `id` 属性和属性别名

前面提到过 `id` 属性，其实它就是一个对象的名字，来唯一确定一个对象，在其他对象中可以通过 `id` 引用该对象。下面我们将 `MyHelloWorld.qml` 文件的内容更改如下：

```
import QtQuick 2.0
Rectangle {
    width: 100
    height: 62
    property alias mArea: mouseArea
    Text {
       text: "hello World"
       anchors.centerIn: parent
    }
    MouseArea {
       id: mouseArea
       anchors.fill: parent
    }
}
```

这里我们首先更改了 `MouseArea` 对象，设置其 `id` 为 `mouseArea` ，这样就可以在 `Rectangle` 中通过 `mouseArea` 来访问 `MouseArea` 对象。为了可以在 `MyHelloWorld.qml` 文件外访问 `Rectangle` 内的子对象，我们需要在 `Rectangle` 中自定义属性，并且该属性需要是子对象的属性别名，例如这里我们声明了一个 `mArea` 属性， `alias` 表明 `mArea` 是 `mouseArea` 的别名，这样在 `MyHelloWorld.qml` 文件外就可以通过 `mArea` 来操作 `MouseArea` 对象了。

下面我们修改 `main.qml` 文件：

```
import QtQuick 2.0
Item {
    MyHelloWorld {
       anchors.fill: parent

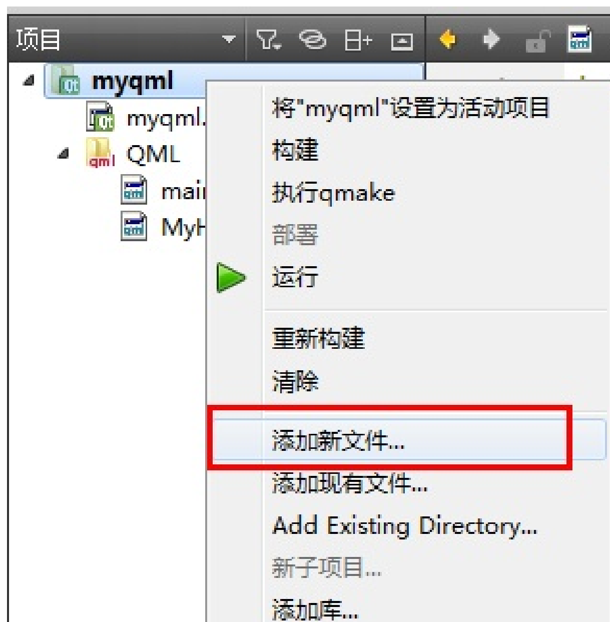
       mArea.onClicked: {
           Qt.quit()
       }
    }
}
```

这里直接在 `MyHelloWorld` 对象中通过 `mArea` 调用了 `onClicked` 处理器。现在保存文件，然后通过 `qmlscene` 运行程序，查看效果。

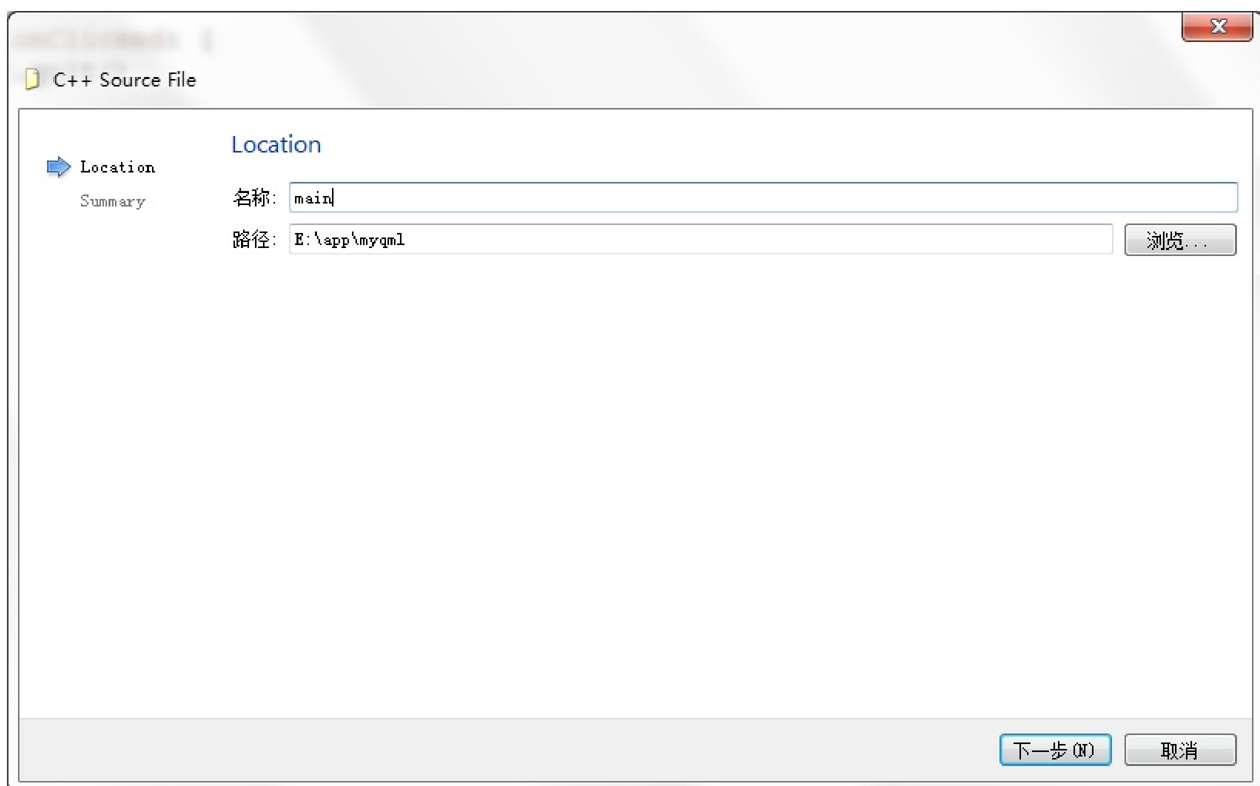
到这里我们已经基本还原了上一篇中创建的Qt Quick应用中 `qml` 文件的内容，从最简单的程序开始，逐渐丰富代码，现在大家应该已经对QML程序有了一定的认识。下面我们继续丰富程序，使其成为一个可以编译运行的项目。

五、添加C++代码

1、添加 `main.cpp` 文件。首先在 `myqml` 目录上右击，选择“添加新文件”，如下图所示。在弹出的对话框中选择C++分类中的“C++Source File”。



2、文件名称设置为 `main` ，如下图所示。



3、添加完成后将 `main.cpp` 文件的内容修改为：

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("../myqml/main.qml")));
    return app.exec();
}
```

这里主要就是普通的Qt C++中 `main()` 函数，关键是其中创建了 `QQmlApplicationEngine` 对象来加载QML文件，这个是固定的用法。因为现在编译运行程序的本地目录是自动生成的目录，它与 `myqml` 源码目录是同级目录，所以加载 `main.qml` 文件需要指定相对路径。

2、更改项目文件。现在在 `myqml.pro` 文件中已经自动添加了一些代码，我们只需在最后添加一句代码：

```
QT += quick qml
```

这样就表明项目中使用了 `QtQuick` 和 `QtQml` 模块。

3、修改 `main.qml` 文件。要在C++程序中使用 `QQmlApplicationEngine` 加载QML文件，要求顶层窗口使用 `Window` 项目，就是在 `main.qml` 中的根对象使用 `Window` 对象来代替 `Item` 对象，因为 `Window` 默认是不显示的，所以还有设置其 `visible` 属性为 `true` ，这个也是固定用法，大家以后照着做即可。修改后的 `main.qml` 文件如下：

```
import QtQuick 2.0
import QtQuick.Window 2.0
Window {
    visible: true
    MyHelloWorld {
       anchors.fill: parent
       mArea.onClicked: {
           Qt.quit()
       }
    }
}
```

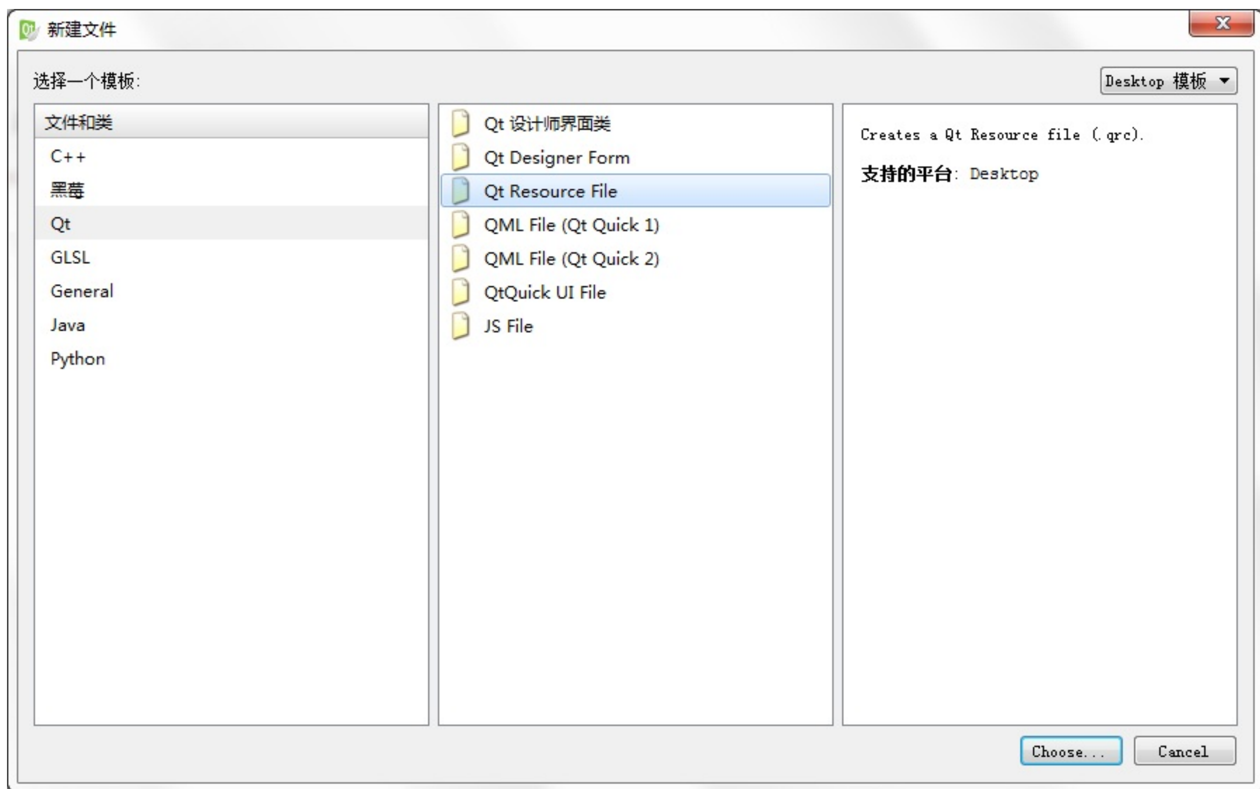
这里还导入了 `QtQuick.Window` 2.0，这是因为该模块包含了 `Window` 类型。现在我们可以按下 `Ctrl+R` 快捷键来编译运行程序了，效果如下图所示。



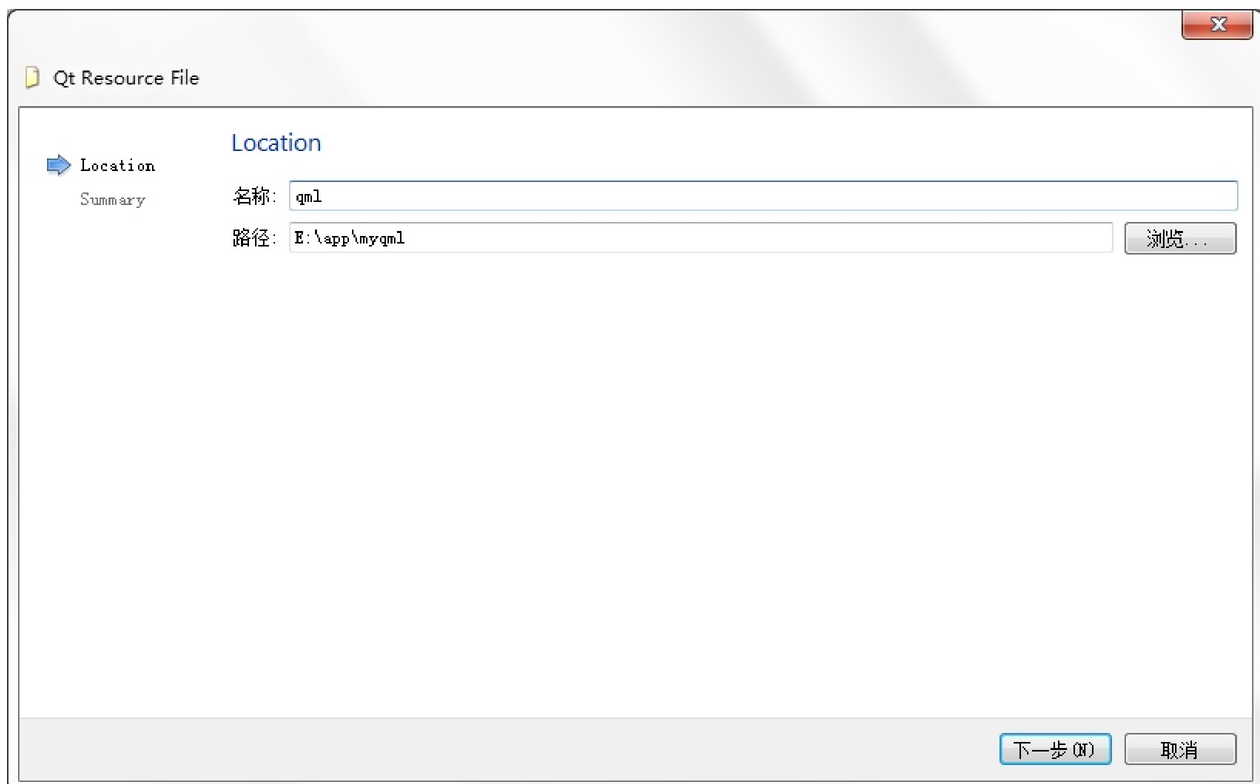
六、使用资源文件

因为QML文件是普通的文本文件，但在程序运行时需要使用这些文件来创建界面，将其直接放在程序外是不安全的，所以我们的做法是将QML文件放到资源文件中。

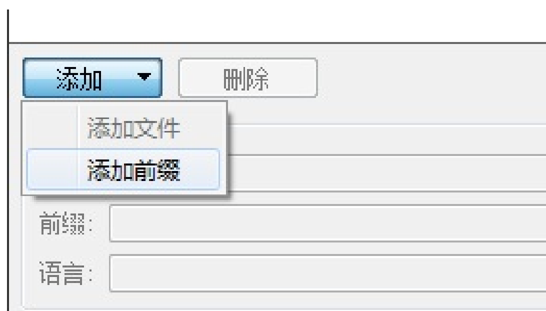
1、添加资源文件。继续在myqml目录上右击，选择添加新文件，模板选择Qt分类中的“Qt Resource File”，如下图所示。



然后将文件名称设置为“qml”，如下图所示。



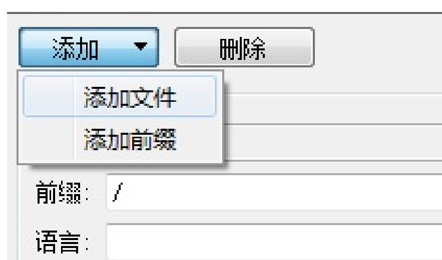
2、文件创建好之后会自动打开，这里我们先点击“添加”按钮，选择“添加前缀”，如下图所示。



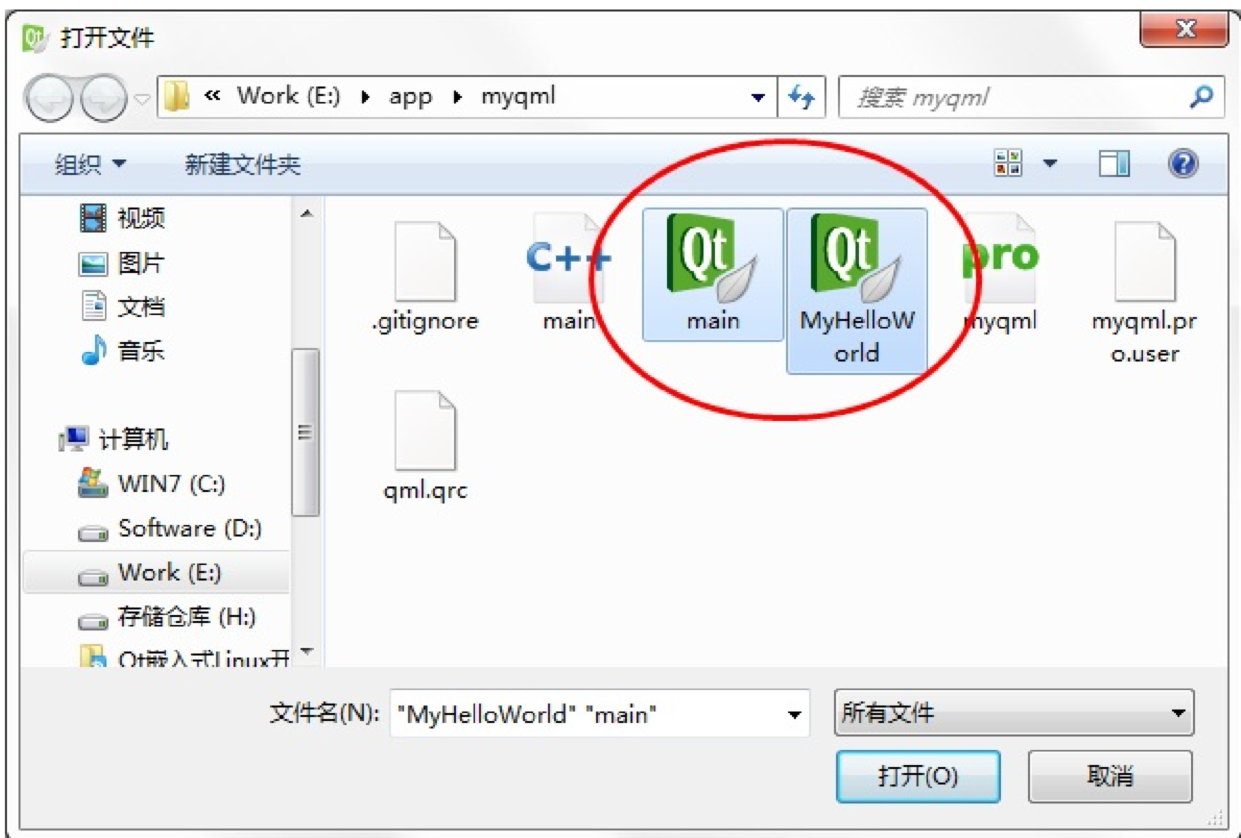
这里将前缀设置为 `/` ，如下图所示。（其实前缀使用什么都可以，这里为了简便，只保留斜杠）



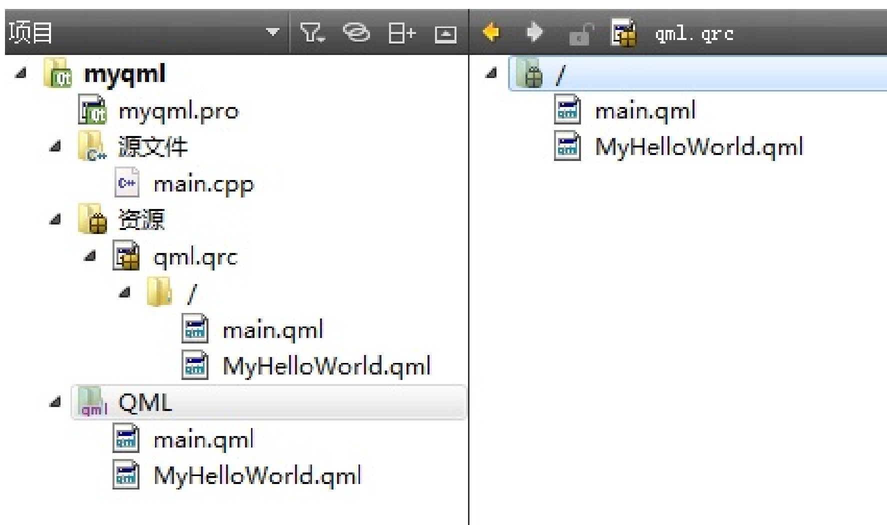
3、再次点击“添加”按钮，这次选择“添加文件”，如下图所示。



在弹出的对话框中，选择源码目录中所有的QML文件，如下图所示。完成后点击 `Ctrl+S` 保存资源文件。



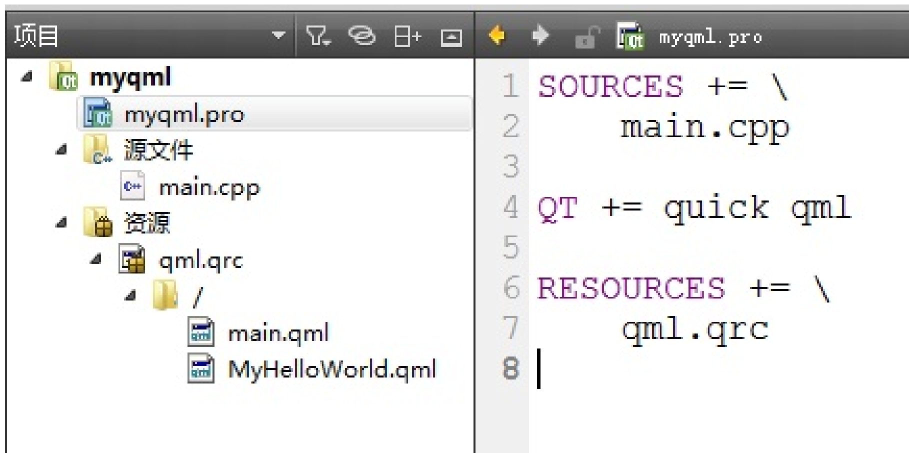
4、添加完成以后，可以在左边文件列表中看到资源文件。如下图所示。



5、因为已经在资源文件中添加了QML文件，所以以前在项目中的QML文件就不再需要了。双击打开 `myqml.pro` 文件， 然后将其中的：

```
DISTFILES += \
    main.qml \
    MyHelloWorld.qml
```

代码删除掉，点击 `Ctrl+S` 保存项目文件。这时项目中的文件列表如下图所示。



6、最后修改`main.cpp`文件。因为现在使用了资源文件，所以要更改`main.cpp`文件中QML文件的路径，将其修改为：

```
engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
```

到这里整个项目就修改完成了，大家可以运行程序，查看效果。这个项目将作为我们的模板，后面的章节会在这个项目的基础上进行修改来讲解，项目的创建过程就不再赘述。

小结

这一篇中我们将上一篇讲解的Qt Quick项目进行拆解分析，以初学者的角度，从最简单的几行代码开始，一点点丰富程序，一步步进行分析讲解，最终还原了整个程序。本节内容让大家可以从根本上掌握Qt Quick程序的构建，并且也提供了一种学习的方法，一种分析大型复杂程序的方法，就是所谓的剖析法，希望大家用心过一遍，为以后的学习打好基础。